A Logical Framework for Template Creation and Information Extraction
— Technical Report —

David Corney[1], Emma Byrne[2], Bernard Buxton[1], and David Jones[1]

October 14, 2005

Abstract

Information extraction is the process of automatically identifying facts of interest from pieces of text, and so transforming free text into a structured database. 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, and in this paper, 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 a fair 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. This paper introduces such a framework which 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. Our framework describes formally what a template is, with definitions of words and documents, and covers other typical information extraction and text mining tasks, such as stemming and part of speech tagging, as well as information extraction itself. We show how common search algorithms could be used to create and optimise templates automatically, using sequences of overlapping templates, and we develop heuristics that make this search feasible.

[1]Department of Computer Science, University College London, Gower Street, London WC1E 6BT
Email: D.Corney@cs.ucl.ac.uk or D.Jones@cs.ucl.ac.uk

[2]The Business School, University of Greenwich, Old Royal Naval College, Park Row, London, SE10 9LS
Email: E.Byrne@greenwich.ac.uk

Contents

1 Introduction **2**

2 Basic definitions **3**
2.1 Membership of terms 5
2.2 Templates and document fragments 6
2.3 Co-occurrence Analysis 6

3 Template ordering **6**
3.1 Superset ordering of terms and template elements 7
3.2 Ordering of templates 8

4 Template generalisation **8**
4.1 Creating and modifying single template elements 8
4.2 Creating and modifying entire templates 10
4.3 Matching fragments of differing lengths 10

5 Measuring template quality **11**
5.1 Defining recall and precision 11

6 Searching for good templates **12**
6.1 Estimating recall and precision 12
6.2 Search algorithms 14
6.3 Feeding knowledge forward 14
6.4 A best-first algorithm 15
6.5 Multi-objective search methods 18

7 Discussion and extensions **18**

8 Conclusion **19**

1 Introduction

Information extraction (IE) [7] has developed over recent decades with applications analysing text from news sources [8], financial sources [6], and biological research papers [1,5,12]. Competitions such as MUC and TREC have been promoted as using real text sources to highlight problems in the real world, and more recently TREC has included a genomics track [11], again highlighting biology and medicine as growing areas of IE research.

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.

The terms information extraction and text mining are often used interchangeably. Some authors use the term text mining to suggest the detection of novelty, such as combining information from several sources to generate and test new hypotheses [24]. In contrast, IE extracts only that which is explicitly stated. This framework focuses on IE and template creation, but it also applies to text mining.

Information extraction is a large and diverse research area. One approach widely used is to develop modular systems such as GATE [9], so that each component can be optimised individually. One of the most challenging of these is the template component. 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. Standard regular expressions match sequences of characters, but IE templates can also match features of words. To guide our discussion, consider the sentence "The cat sat on the mat".

Informally, one template that would match that sentence is “The ANIMAL VERB on the FLOOR_COVERING”, where ‘ANIMAL’ and ‘FLOOR_COVERING’ are predefined semantic categories, and ‘VERB’ is a part-of-speech label. A different template that would match the same sentence is “DETERMINER * * PREPOSITION DETERMINER *”, where each ‘*’ is a wildcard, matching any single word, and the other symbols are part-of-speech labels. 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 [2, 13, 19], 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 [22], which is also laborious.

One way to view the automatic creation of useful templates is as a search problem of a kind familiar to the artificial intelligence community [23, ch. 3–4]. To formulate it this way, we need to 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). Any useful framework describing IE must provide a way to define and create templates, and our framework proposes using these AI search methods, an idea we expand in Section 6.2, where we “grow” useful templates from given seed phrases.

One alternative to using templates is *co-occurrence analysis* [16]. 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 2.3.

The framework itself is presented in Sections 2–5, with the subsequent sections discussing various implementation issues.

Section 2 defines various concepts formally, moving from words and documents to templates and information extraction. Section 3 describes how templates can be ordered according to how specific or general they are, as a precursor to template creation and optimisation. Section 4 discusses how to modify a template to make it more general. Section 5 gives formal definitions of recall and precision within our framework and discusses how they might be estimated in practice. Section 6.2 discusses heuristic search algorithms and their feasibility, before a concluding discussion.

A shorter form of this work is published in [4].

2 Basic definitions

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

Definition 2.1. *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 2.2. *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 2.3. *A* corpus D *is a set of documents:* $D = \{d_1, d_2, \ldots, d_{|D|}\}$.

Example: $D_1 = \{d_1, d_2\}$.

Definition 2.4. *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: Λ_{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, 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.

Words that share a common stem, or root, typically share a common meaning, such as the words "sit", "sitting" and "sits". It is therefore common practice in information retrieval to index words according to their stem to improve the performance [21]. Similarly in information extraction, it is often helpful to identify words that share a common stem. The most common approach is to remove suffixes to produce a single stem for each word [21], although in principle, each word could have multiple stems, such as if prefixes were removed independently of suffixes.

Words may also belong to pre-defined semantic categories, such as "business", "country" or "protein". One common way to define these semantic categories is by using gazetteers. A gazetteer is a named list of words and phrases that belong to the same category. Rather than simple lists, some ontologies are based on hierarchies or directed acyclic graphs, such as MeSH[1] and GO[2] respectively. 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. Common examples are noun, verb and adjective, although these are often subdivided into more precise categories such as "singular common noun", "plural common noun", "past tense verb" and so on. The part of speech can usually only be ascertained for a word in a given context. For example, compare "He cut the bread" to "The cut was deep". In practice, an implementation may limit this to exactly one label per word, based on the context of that word. Following the Penn Treebank tags [17], in some examples 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.

We also introduce wildcards as an extension to the idea of word attributes. 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. Then 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 4.3.

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 2.5. *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. This 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"; Γ for "gazetteers"; Σ for "stems"; and Ω for "wildcards".

Example:
$\kappa_\Lambda = \{$the, cat, sat, on, mat, mouse, ...$\}$
$\kappa_\Sigma = \{$the_stem, cat_stem, sit_stem, on_stem, mat_stem, mouse_stem, ...$\}$
$\kappa_\Pi = \{$DT, NN, VBD, IN, ...$\}$
$\kappa_\Gamma = \{$FELINE, RODENT, ANIMAL, FLOOR_COVERING, ...$\}$
$\kappa_\Omega = \{*, ?\}$
We use the suffix '_stem' in stem labels to avoid confusing them with the corresponding literal.

Definition 2.6. *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 2.7. *A term t is a value that an attribute may take, i.e. an element of a category of word attributes.*

[1]MeSH, Medical Subject Headings, `http://www.nlm.nih.gov/mesh`
[2]Gene Ontology, `http://www.geneontology.org`

Examples: $t_1 = \text{cat}$, $t_2 = \text{NN}$, $t_3 = \text{FELINE}$, where $t_1 \in \kappa_\Lambda$, $t_2 \in \kappa_\Pi$, $t_3 \in \kappa_\Gamma$.

Definition 2.8. *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:
$T_1 = \{\text{NN, VBD}\}$
$T_2 = \{\text{FELINE, RODENT, FLOOR_COVERING}\}$
The set {NN, FELINE } is not a template element because "NN" and "FELINE" belong to different categories, namely κ_Π and κ_Γ respectively.

The name "template element" refers to templates as defined in Definition 2.13 below.

Definition 2.9. *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* $\alpha(\lambda, \kappa) = \{T | \forall t \in T,\ t \in \kappa \text{ and } \lambda \text{ has attribute } t\}$*. 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 maybe functions to assign a stem attribute to a word, or to assign a particular semantic category to any of a given list of words.

For convenience, we label the attributes using the category label as a subscript in these examples.

Examples:
$\alpha_\Lambda(\text{cat}) = \{\text{cat}\}$
$\alpha_\Gamma(\text{cat}) = \{\text{FELINE, ANIMAL}\}$
$\alpha_\Pi(\text{cat}) = \{\text{NN}\}$
$\alpha_\Sigma(\text{cat}) = \{\text{cat_stem}\}$
$\alpha(\text{cat}) = \{\{\text{cat}\}, \{\text{NN}\}, \{\text{FELINE, ANIMAL}\}, \{\text{cat_stem}\}\}$
In the case of $\alpha(\text{the})$, the word "the" has a part-of-speech tag "DT" (determiner) and the obvious literal, but no gazetteer or stem entries. So $\alpha_\Pi(\text{the}) = \{\text{DT}\}$, and $\alpha_\Gamma(\text{the})$ 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, $|\alpha_\Lambda(\lambda)| = 1$. As a consequence, $|\alpha(\lambda)| \geq 1$.

2.1 Membership of terms

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

Definition 2.10. *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.* $\mu(t) = \{\lambda | t \in \alpha(\lambda)\}$*. 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:
$\mu(\text{sit_stem}) = \{\text{sit, sits, sat, sitting}\}$.
$\mu(\text{FELINE}) = \{\text{cat, lion, tiger}, \ldots\}$.
$\mu(\text{RODENT}) = \{\text{mouse, rat, hamster}, \ldots\}$.
$\mu(\{\text{FELINE}, \text{RODENT}\}) = \{\text{cat, lion, tiger, mouse, rat, hamster}, \ldots\}$.
$\mu(\alpha_\Lambda(\text{cat})) = \{\text{cat}\}$. I.e. the membership of the literal category of a literal is the set containing only the literal itself.

Definition 2.11. *We also define membership for a term limited to a particular document or set of documents:* $\mu(t, d) = \{\lambda | \lambda \in d, \lambda \in \mu(t)\}$*, and* $\mu(t, D) = \bigcup_{d \in D} \mu(t, d)$

Examples:
$\mu(\text{NN}, d_1) = \{\text{cat, mat}\}$.
$\mu(\text{ANIMAL}, D_1) = \{\text{cat, mouse}\}$.

2.2 Templates and document fragments

Definition 2.12. *We define a* fragment *of a document as being a tuple of successive literals taken from some document d. If*

$$d = < \lambda_1, \lambda_2, \ldots, \lambda_a, \lambda_{a+1}, \ldots, \lambda_{a+b-1}, \ldots, \lambda_{|d|} >,$$

then

$$f(d, a, b) = < \lambda_a, \lambda_{a+1}, \ldots, \lambda_{a+b-1} > .$$

I.e. the function $f(d, a, b)$ returns a tuple of b words in order, from d, starting with the ath word. f_1 is the first word of the fragment, i.e. λ_a, f_2 is the second word, λ_{a+1}, and so on. Note that $|f| = b$.

Example: If d_1 =<the, cat, sat, on, the, mat >, then $f(d_1, 4, 3)$ =<on, the, mat>.

Definition 2.13. *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 2.14. *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. Let $f = < f_1, \ldots, f_n >$ be a fragment. Given 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 template $\tau_1 = <$ {DT}, {ANIMAL}, {VBD} $>$ and a corpus D_1 (as defined after Definition 2.3), then $\mu(\tau_1, D_1)$ = {<the, cat, sat>, <a, mouse, ran>}.

2.3 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 containing words that match our terms. With three or more entities, larger sets of templates may be required.

3 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.

For any given document, each template matches a certain number of fragments. A template that matches every possible fragment is useless, as is one that matches no fragments at all. Somewhere between these two extremes of generic templates and

specific templates, 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.

In this section, we define possible orderings of terms and templates. In the next section, we define algorithms that use these orderings to modify templates. In section 6.2 we discuss some search algorithms that might be used to locate optimal templates.

Before we consider an ordering over templates, we consider ordering over the terms that make up a template. We want to define the relations $t_1 >_s t_2$ to mean that term t_1 is a more specific attribute than term t_2, and $t_1 \geq_s t_2$ to mean that t_1 is at least as specific as t_2. Similarly, $t_1 <_s t_2$ and $t_1 \leq_s t_2$ mean "less specific than" and "no more specific than" respectively. We now define these by introducing "superset ordering".

3.1 Superset ordering of terms and template elements

We start by defining 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 3.1. *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.*

Examples:

Let FELINE and ANIMAL be two terms (specifically, gazetteers), such that $\mu(\text{FELINE}) = \{\text{cat, lion, tiger , } \ldots\}$ and $\mu(\text{ANIMAL}) = \{\text{antelope, dog, cat, } \ldots, \text{lion,} \ldots \text{,tiger,} \ldots \text{,zebra}\}$. Then $\mu(\text{FELINE}) \subset \mu(\text{ANIMAL})$ and so FELINE $>_s$ ANIMAL.

$\mu(\alpha_\Lambda(\text{cat})) = \{\text{cat}\}$.

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

$\mu(\{\text{FELINE}, \text{ANIMAL}\}) = \{\text{cat, lion, tiger, antelope, dog, } \ldots\}$

Therefore $\mu(\alpha_\Lambda(\text{cat})) \subset \mu(\alpha_\Gamma(\text{cat}))$.

Some specificity orderings are dependent on the categories of the two terms. For example, by definitions 2.9 and 2.10, it is clear that each literal is a member of all the terms that are attributes of the literal. Therefore $\mu(\lambda) \subseteq \mu(\alpha_\lambda) \forall \kappa \in K$ and therefore 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. At the other extreme, 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 $\forall\, t_2 \in K,\ t_2 \geq_s t_1$.

Having defined an ordering over terms, we now consider sets of terms, and template elements in particular. 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:

$$T_1 \geq_s T_2 \iff \forall \lambda \in \mu(T_1),\ \lambda \in \mu(T_2)$$

Similarly,

$$T_1 >_s T_2 \iff \forall \lambda \in \mu(T_1),\ \lambda \in \mu(T_2) \text{ and } \exists \lambda \in \mu(T_2) \text{ such that } \lambda \notin \mu(T_1).$$

I.e. 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.

We can trivially extend these 'more specific than' relations to define 'less specific than' and 'equally specific as' relations:

$$t_1 >_s t_2 \iff t_2 <_s t_1$$
$$t_1 \geq_s t_2 \iff t_2 \leq_s t_1$$
$$t_1 =_s t_2 \iff t_1 \geq_s t_2 \text{ and } t_1 \leq_s t_2$$

In this last case, $t_1 =_s t_2 \iff \mu(t_1) = \mu(t_2)$. Note that this is a narrower definition than requiring that $|\mu(t_1)| = |\mu(t_2)|$.

3.2 Ordering of templates

Above, 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 generalisation, then for a given set of documents D, this is testing whether $\mu(\tau_1, D) \supset \mu(\tau_2, D)$ and therefore whether $|\mu(\tau_1, D)| > |\mu(\tau_2, D)|$, or vice versa, or they are equal. E.g. $\tau_1 >_s \tau_2 \iff \mu(\tau_1, D) \subset \mu(\tau_2, D)$. This depends on the corpus D and is potentially slow to evaluate, especially if D contains a large number of documents. We would rather have an estimate of the relative specificity which is independent of D, which will be useful when developing search heuristics.

Suppose τ_1 and τ_2 are almost identical, and differ only in one template element:

$$\tau_1 =< T_1, T_2, \ldots, T_i, \ldots, T_n >$$

$$\tau_2 =< T_1, T_2, \ldots, T_i', \ldots, T_n >$$

where $T_i \neq T_i'$. Then we can say that $\tau_1 >_s \tau_2 \iff T_i >_s T_i'$.

More generally, if τ_1 and τ_2 contain the same number of sets of terms, then

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

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

In a slight extension to previous notation, we use $T_{n,i}$ to refer to the ith 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 compare similar templates, and this is sufficient to 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 4, and to develop methods to search efficiently for *good* templates in Section 6.2.

4 Template generalisation

Whether created manually or automatically, templates are usually based on examples of "interesting" phrases. These phrases may be identified by hand (e.g. by a domain expert) or automatically (e.g. by information retrieval methods). These phrases are then used as "seeds" to help define more general templates. In this framework we will follow this "seed phrase" approach, and assume that we have 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 (Section 4.2).

4.1 Creating and modifying single template elements

We now bring together several concepts discussed above, and define functions that create and generalise template elements. This leads onto a discussion of creating and modifying entire templates.

Definition 4.1. *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:* 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 3.1) 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 4.2. *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* $\{T'_1, T'_2, \ldots, T'_{|T'|}\}$ *:*

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

I.e. For each template element produced by $\textsf{generalise}(T, \kappa')$, 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 ensures that the new element is *more* general than the original, and not just *as* general.

Example: Let category κ_Γ = {FELINE, CANINE, ANIMAL} contain three sets of literals:
$\mu(FELINE)$ = {cat},
$\mu(CANINE)$ = {dog}, and
$\mu(ANIMAL)$ = {cat, dog, mouse, horse}.

Let $T = \{t_1, .., t_n\}$ such that $\bigcup_{i=1}^{m} \mu(t_i)$ = {cat, dog}. Then $|\mu(FELINE) \cap \bigcup_{i=1}^{m} \mu(t_i)| = 1$, therefore the set $FELINE$ can be considered for inclusion in the sets in $\textsf{generalise}(T, \kappa_\Gamma)$. $|\mu(CANINE) \cap \bigcup_{i=1}^{m} \mu(t_i)| = 1$, therefore the set $CANINE$ can be considered for inclusion in the sets in $\textsf{generalise}(T, \kappa_\Gamma)$. $|\mu(ANIMAL) \cap \bigcup_{i=1}^{m} \mu(t_i)| = 2$, therefore the set $ANIMAL$ can be considered for inclusion in the sets in $\textsf{generalise}(T, \kappa_\Gamma)$. The sets $\{FELINE\}$ and $\{CANINE\}$ are individually insufficient to represent all the literal members of $\textsf{generalise}(T, \kappa_\Gamma)$ and so will not be in $\textsf{generalise}(T, \kappa_\Gamma)$. The remaining candidate sets are

$$\{\{\text{ANIMAL}\},$$
$$\{\text{FELINE, CANINE}\}, \{\text{FELINE, ANIMAL}\}, \{\text{CANINE, ANIMAL}\},$$
$$\{\text{FELINE, CANINE, ANIMAL}\}\}.$$

Note that {ANIMAL} ⊂ {FELINE, ANIMAL}, {ANIMAL} ⊂ {CANINE, ANIMAL}, and {ANIMAL} ⊂ {FELINE, CANINE, ANIMAL}. Likewise {FELINE, CANINE} ⊂ {FELINE, CANINE, ANIMAL}. As a result, {FELINE, ANIMAL}, {CANINE, ANIMAL} and {FELINE, CANINE, ANIMAL} are excluded from $\textsf{generalise}(T, \kappa_\Gamma)$, because there are subsets of these sets that meet the conditions of generalise. Therefore $\textsf{generalise}(T, \kappa_\Gamma)$ = {{FELINE, CANINE}, {ANIMAL}}. This is a set of two template elements, the first consisting of two terms and the second consisting of one term.

Note that the cardinality of the sets returned by $\textsf{generalise}(T, \kappa_\Gamma)$ is not necessarily an indication of their specificity. For example, the set {ANIMAL} matches more literals but represents a single semantic category, while the set {FELINE, CANINE} matches fewer literals but combines two semantic categories. In an implementation we may decide to use the set {FELINE, CANINE} which is a more cumbersome, but more specific set of terms than {ANIMAL}, depending on our requirements. The decision made regarding which set to use in a template would depend on the application and on the details of the heuristic searches, as we discuss in Section 6.2.

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 $\textsf{generalise}(T, \kappa_x) = \emptyset$.

Example: $\text{generalise}(\{\text{the}\}, \kappa_\Gamma) = \emptyset$, because the literal "the" is not in any gazetteer.

4.2 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 2.13), 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 4.3. *We extend the* initialise *function to create a specialised template from a fragment.*

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

This can also be written as: $\textit{initialise}(f) = f$*, where* f *is a text fragment.*

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 4.1. One template will be created for each possible generalisation of the specified template element.

Definition 4.4. *Given the template,* $\tau =< T_1, T_2, \ldots, T_i, \ldots, T_n >$*. Then*

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

I.e. we replace the ith template element with the result of its own generalisation.

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

4.3 Matching fragments of differing lengths

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. This is clearly undesirable, so we will now extend the framework to allow for templates that match fragments of various lengths, firstly to shorter fragments and then to longer ones.

Suppose we want to be able to define a template that can match a particular class of noun phrase, in the form of a determiner, zero or more words (e.g. adjectives), and an animal. To do this, we use the wildcard attribute '?' which matches any single word *or no word at all.* This can be used to define templates such as $\tau =<$ DT, ?, ?, ANIMAL $>$.

To use the '?' wildcard in our examples, we assume that '?' is in the wildcard category κ_Ω and use the existing $\text{generalise}(T, \kappa_\Omega)$ function. So for example, we take a fragment and apply a series of generalise functions after initialisation like this:

$$\begin{aligned}
\tau_1 &=< \text{the, quick, brown, fox} > && = \text{initialise}(\text{the, quick, brown, fox})\\
\tau_2 &=< \text{the, ?, brown, fox} > && \in \text{generalise}(\tau_1, \kappa_\Omega, 2)\\
\tau_3 &=< \text{the, ?, ?, fox} > && \in \text{generalise}(\tau_2, \kappa_\Omega, 3)\\
\tau_4 &=< \text{DT, ?, ?, fox} > && \in \text{generalise}(\tau_3, \kappa_\Pi, 1)\\
\tau_5 &=< \text{DT, ?, ?, ANIMAL} > && \in \text{generalise}(\tau_4, \kappa_\Gamma, 4)
\end{aligned}$$

Consider a document containing these fragments:

$$\begin{aligned}
f_1 &=< \text{the, cat} >\\
f_2 &=< \text{the, lazy, dog} >\\
f_3 &=< \text{the, quick, brown, fox} >
\end{aligned}$$

Of the above list of templates, τ_5 matches all three fragments, whereas $\tau' =$<DT, *, *, ANIMAL> would not match f_1 or f_2.

Although this approach allows templates to match fragments of varying length as desired, it is restricted to matching fragments that are no longer than the original seed. The seed phrase must therefore be chosen with this in mind, or else modified before the search begins by inserting wildcards.

One solution is to extend the generalise function so that it inserts a wildcard into the template, making the template longer and allowing it to match phrases longer than the seed phrase. For example, we could define generalise$(\tau, \kappa_{\Omega'}, i)$ as a function that creates new templates from template τ by inserting wildcards from $\kappa_{\Omega'}$ between template elements i and $i+1$. The category $\kappa_{\Omega'}$ contains the same wildcards as κ_Ω, with the distinction that they are inserted *after* existing template elements, rather than used to replace template elements. Introducing extra wildcards in this way complicates the search algorithms substantially, so restrictions would have to be introduced, either limiting how many wildcards could be added to a template, or limiting the maximum length of a template. These heuristics could be chosen for a particular implementation after experimentation.

Note that there can be no gain from inserting '?' wildcards before the first non-wildcard template element, or after the last non-wildcard element, as the resulting template would match exactly the same fragments with or without these optional 'external' wildcards.

5 Measuring template quality

In this section, we define recall, precision and related terms formally within our framework. We then discuss how these can be estimated in practice. These measures are needed to guide the automatic search for templates discussed in the next section.

5.1 Defining recall and precision

We have already defined a function $\mu(\tau, D)$ that retrieves the set of document fragments that are matched by template τ from document set D (Definition 2.14).

Definition 5.1. *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[3], but allows us to define concepts such as "true positive" and the precision rate. Figure 1 shows the relationships diagrammatically. We use the notation $\wp_f(D)$ to show the set of all fragments in D, independent of any template. This is the set of all tuples of all lengths obtainable as fragments using $f(d, a, b)$ (Definition 2.12).

We now define a series of sets with respect to a template τ and a corpus D.

Definition 5.2. *True-positives* $TP(\tau, D) = \mu(\tau, D) \cap I(D)$

Definition 5.3. *False-positives* $FP(\tau, D) = \mu(\tau, D) \setminus I(D)$

Definition 5.4. *False-negatives* $FN(\tau, D) = I(D) \setminus \mu(\tau, D)$

Definition 5.5. *True-negatives* $TN(\tau, D) = D \setminus \{I(D) \cup \mu(\tau, D)\}$

Definition 5.6. *Recall* $r(\tau, D) = \frac{|\mu(\tau,D) \cap I(D)|}{|I(D)|} = \frac{|TP|}{|I(D)|}$

Definition 5.7. *Precision* $p(\tau, D) = \frac{|\mu(\tau,D) \cap I(D)|}{|\mu(\tau,D)|} = \frac{|TP|}{|\mu(\tau,D)|}$

We can conceive of a "perfect" template, τ^*, which matches all the positive fragments, and nothing else. I.e. $\mu(\tau^*, D) = I(D)$. This has a recall and a precision of one. Although such a perfect template may not be known in general, we can use these definitions of true positive and false positive counts to guide a search for templates.

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

[3] This set is task-dependent and user-dependent, as different people will pick different fragments as being interesting, even if attempting the same task [3].

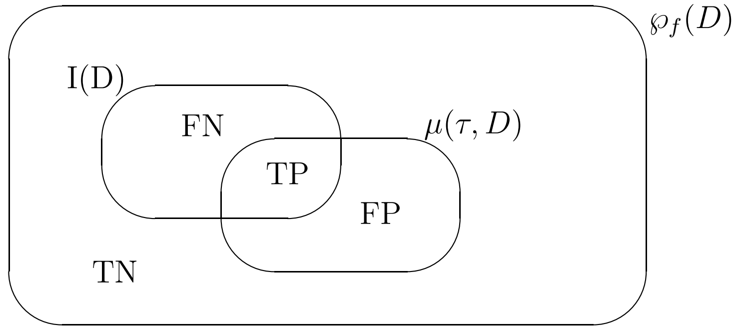


Figure 1: Venn diagram showing the relationship between: the set of all possible fragments $\wp_f(D)$; the results of applying template τ to it — match(τ, D); the ideal set of results $I(D)$; and the false negative (FN), true positive (TP), false positive (FP) and true negative (TN) regions.

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 [10], as we discuss further in Section 6.5.

6 Searching for good templates

In 6.2 we will discuss the development of search algorithms, but first we need a practical way to estimate recall and precision.

6.1 Estimating recall and precision

As noted earlier, we do not know which fragments are interesting *a priori*, and so the above definitions 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. We now consider several options.

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. [14]), 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 one set of documents where for each sentence, the probability that it contains a relevant piece of information is above some threshold, and a second set of documents, where the probability is below a threshold. We could then treat this as a classification problem, albeit with noisy labels on the data. However, such a set of irrelevant documents is hard to define, and furthermore, even interesting documents are likely to contain irrelevant facts, such as background information.

Suppose that instead of having irrelevant (negative) documents, we have 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. 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.

Let D be a corpus containing $|D|$ documents. We define the set of positive documents as D^+ and neutral documents as D^N, such that $D = D^+ \cup D^N$ and $D^+ \cap D^N = \emptyset$. A "positive" document is one that the user believes is likely to contain information of interest. A "neutral" document is one where the user has no reason to believe that the document does or does not contain information of interest. We now use these two sets of documents to define estimates of the numbers of true-positive fragments and false-positive fragments matched by a template τ.

Definition 6.1. *We define an estimated true-positive set* $\widehat{TP}(\tau, D) = \mu(\tau, D^+)$*, for a template* τ *and a set of positive documents,* $D^+ \subseteq D$.

Definition 6.2. *We define an estimated false-positive set* $\widehat{FP}(\tau, D) = \mu(\tau, D^N)$*, for a template* τ *and a set of neutral documents,* $D^N \subseteq D$.

These are very crude estimates, as they assume that every fragment matched by τ in D^+ contains information of interest, and that every fragment matched by τ in D^N contains *no* information of interest. Thus they cannot be used to estimate the precision or recall scores[4], but they are sufficient to guide the search of useful templates. In fact, as a successful search progresses and the quality of the template improves, then these estimates will become more accurate, although they are unreliable, especially at the start of the search process.

Let us consider some other properties of templates generated using superset generalisation. Suppose we have two templates, τ_1 and τ_2, and that $\tau_1 >_s \tau_2$. Then by definition, $|\mu(\tau_1, D)| < |\mu(\tau_2, D)|$. If τ_2 is a generalisation of τ_1 derived using superset generalisation, 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:

$$|\mu(\tau_1, D^a)| < |\mu(\tau_2, D^a)| \iff |\mu(\tau_1, D^b)| < |\mu(\tau_2, D^b)|.$$

This property will be useful in developing heuristic search methods, because it allows us to rationally prune search graphs, as we discuss in Section 6.2.

We defined terms such as "true positive" and "false positive" above. Now we can say that if template τ_1 is at least as specific as template τ_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:

$$|\mathrm{TP}(\tau_1, D)| \le |\mathrm{TP}(\tau_2, D)|.$$
$$|\mathrm{FP}(\tau_1, D)| \le |\mathrm{FP}(\tau_2, D)|.$$
$$|\mathrm{TN}(\tau_1, D)| \ge |\mathrm{TN}(\tau_2, D)|.$$
$$|\mathrm{FN}(\tau_1, D)| \ge |\mathrm{FN}(\tau_2, D)|.$$

The equivalent inequalities also hold for the estimates defined above:

$$|\widehat{\mathrm{TP}}(\tau_1, D)| \le |\widehat{\mathrm{TP}}(\tau_2, D)|.$$
$$|\widehat{\mathrm{FP}}(\tau_1, D)| \le |\widehat{\mathrm{FP}}(\tau_2, D)|.$$

As these relationships hold for any set of documents D, we can predict some properties of templates without fully evaluating them. We can use these properties to guide heuristic searches.

If our assumptions here are correct, then the probability of finding an interesting fragment is higher in positive documents than in neutral documents. We can write this assumption as $p\left(f \in I(D) | f \in \mu(\tau, D^+)\right) > p\left(f \in I(D) | f \in \mu(\tau, D^N)\right)$.

When comparing two templates, we can say that one is certainly better than the other if either: a) it matches more true positive fragments and no more false positives; or b) it matches fewer false positive fragments and no fewer true positives. Unfortunately, we cannot be certain of whether it is better when c) it matches more true positives and more false positives. We need some way of exploring this trade-off, an issue we return to later.

We can now generate a series of templates of varying generality and compare them with each other in order to guide our search for useful templates. We discuss this further in the following section.

[4]Substitution into definitions 5.6 and 5.7 gives recall $\equiv$ precision $\equiv 1$ for every template, which is clearly optimistic.

6.2 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 2.13) and means to generate (Section 4) and evaluate them (using estimates of true positive and false positive scores given in Section 5, Definitions 6.1–6.2). We now turn to the search algorithms themselves. First, we define an exhaustive search over all possible templates, given a particular seed phrase. We will then show that in general, this is computationally infeasible, owing to the combinatorial growth of the search space with respect to the length of the template. We then introduce heuristics to make the search feasible while still (we hope) producing good templates, and define and discuss a simple best-first search algorithm. We also discuss possible merits of evolutionary algorithms.

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 4.3). From this, we can modify each element in the template by a single application of the $\mathsf{generalise}(\tau, \kappa, x)$ function (Definition 4.4). 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 $\mathsf{generalise}(\tau, \kappa, x)$ function will return 0, 1 or more templates (see Definition 4.2 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. We need some fixed order over the categories, but the actual order is not critical here[5]. For simplicity, let us assume that we have an implementation with five categories, where every word has one literal, one stem, one gazetteer, one part of speech, and one wildcard. We can then list all possible templates derived from a two-term phrase, starting with two sets of literals, and ending with two sets of wildcards:

$< \lambda, \lambda > < \sigma, \lambda > < \gamma, \lambda > < \pi, \lambda > < \omega, \lambda >$
$< \lambda, \sigma > < \sigma, \sigma > < \gamma, \sigma > < \pi, \sigma > < \omega, \sigma >$
$< \lambda, \gamma > < \sigma, \gamma > < \gamma, \gamma > < \pi, \gamma > < \omega, \gamma >$
$< \lambda, \pi > < \sigma, \pi > < \gamma, \pi > < \pi, \pi > < \omega, \pi >$
$< \lambda, \omega > < \sigma, \omega > < \gamma, \omega > < \pi, \omega > < \omega, \omega >$

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), some will have more, and all may have more than one wildcard. We therefore need to introduce heuristics to make the search feasible, unless the seed fragment is very short.

6.3 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 templates will match a superset of the fragments matched by the ancestor template (Section 6.1).

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.

[5]One example would be to use the order: literals, stems, gazetteers, parts-of-speech and wildcards. This order reflects the typical size of the membership functions. E.g. each stem has only a few members; gazetteers often contain hundreds or thousands; parts-of-speech such as verb or noun contain millions of members and so on.

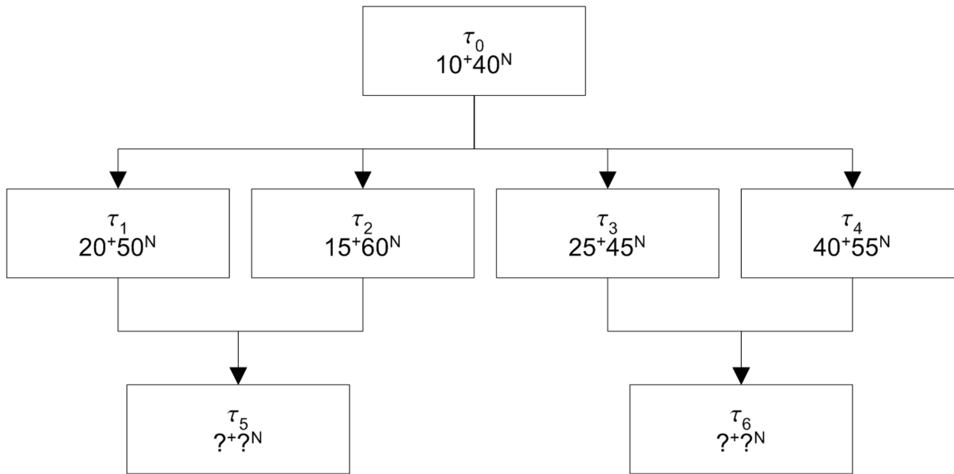


Figure 2: Part of a search tree. Each node is a template with the number of true positives (x^+) and false positives (y^N) shown, with a "?" for unknown values. Each child node can be created by superset generalisation from any of its parents.

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 2. 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.

6.4 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 3 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**

Figure 3: A best-first algorithm. C is the "closed" set of evaluated templates and O is the "open" set of unevaluated templates. See Section 6.4 for further details.

By "best template" (Figure 3, 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).

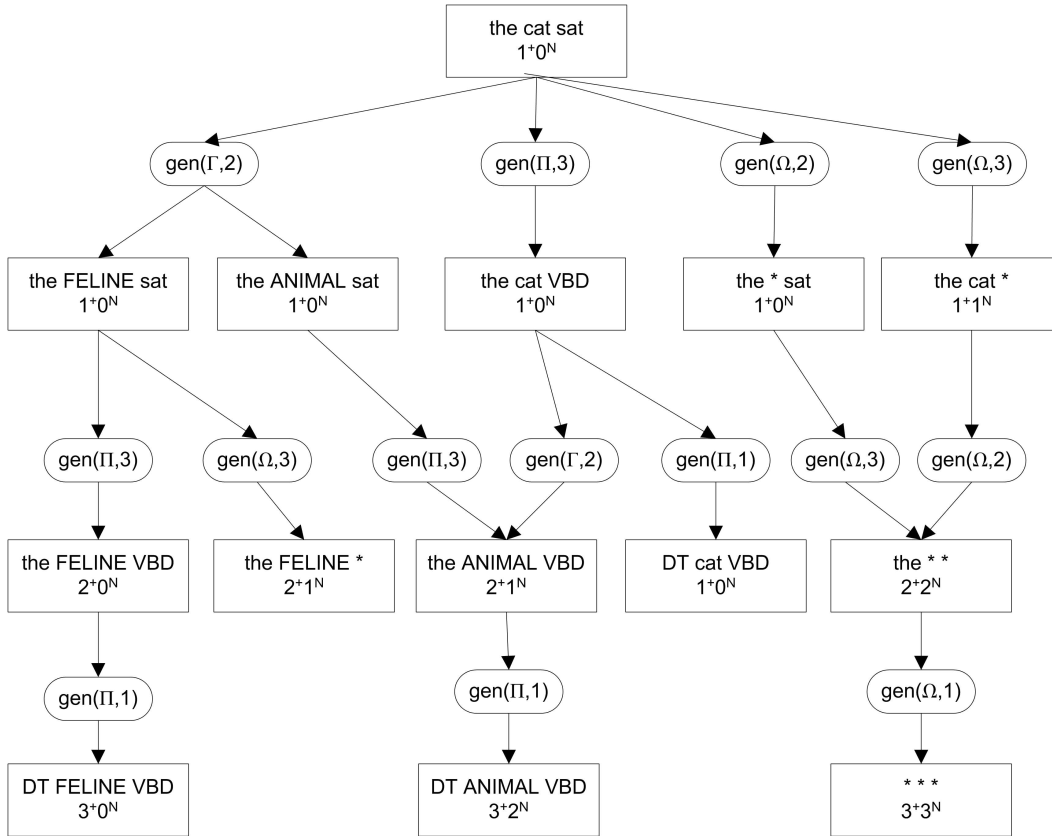
In Step 4f we could choose to either update the bounds of just the descendents of τ in O, or else update the bounds of the descendents of every template in C. The latter would be more computationally expensive, but would lead to better estimates of the lower bounds. For each new template created, it 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 4 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, x) 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 4, i.e. those created after two generalisation functions. From the left of the graph, three of these have two true positives



Legend:

D^+	D^N
{the cat sat}	{the mouse ran}
{the lion roared}	{a cow jumped}
{a tiger slept}	{the cat plays}

Template definition
$TP^{+}FP^{N}$

genaralise(category, position)

Figure 4: 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.

(<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[6], recent variations may provide useful pruning methods, such as SMA* [23, p. 104] and Sweep A* [25].

6.5 Multi-objective search methods

The best-first search described above 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 [10], including problems where evaluations must be limited due to time or financial constraints [15]. 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.

7 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 $\tau_1 =<\{which\}, \{*\}, \{*\}>$, the template $\tau_2 =< \{DT\}, \{ANIMAL\}, \{?^{\tau_1}\}, \{sat\} >$ would match the fragment <the, cat, which, was, black, sat> as well as <the, cat, sat>.

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

An additional approach to finding good templates is to repeatedly *merge* useful templates to produce more general templates [18]. 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 [20]. 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.

8 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, and will allow automatic template creation, and also hope that the framework will allow other researchers to gain further insights into the theory and practice of information extraction and text mining.

Acknowledgements

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

References

[1] C. Blaschke and A. Valencia. The frame-based module of the SUISEKI information extraction system. *IEEE Intelligent Systems*, 17(2):14–20, Mar. 2002.

[2] R. Collier. *Automatic template creation for information extraction.* PhD thesis, Department of Computer Science, University of Sheffield, 1998.

[3] M. E. Colosimo, A. A. Morgan, A. S. Yeh, J. B. Colombe, and L. Hirschman. Data preparation and interannotator agreement. *BMC Bioinformatics*, 6(Suppl 1), 2005.

[4] D. Corney, E. Byrne, B. Buxton, and D. Jones. A logical framework for template creation and information extraction. In *Foundations of Semantic Oriented Data and Web Mining workshop, part of ICDM2005 (the Fifth IEEE International Conference on Data Mining)*, 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*, 20(17):3206–13, 2004.

[6] M. Costantino. *Financial Information Extraction using pre-defined and user-definable templates in the LOLITA System.* PhD thesis, University of Durham, Department of Computer Science, 1997.

[7] J. Cowie and W. Lehnert. Information extraction. *Communications of the ACM*, 39(1):80–91, 1996.

[8] J. Cowie and Y. Wilks. Information extraction. In R. Dale, H. Moisl, and H. Somers, editors, *Handbook of Natural Language Processing*. Marcel Dekker, New York, 2000.

[9] H. Cunningham. GATE, a General Architecture for Text Engineering. *Computers and the Humanities*, 36(2):223–254, May 2002.

[10] C. M. Fonseca and P. J. Fleming. An overview of evolutionary algorithms in multiobjective optimization. *Evolutionary Computation*, 3(1):1–16, 1995.

[11] W. Hersh, R. Bhuptiraju, L. Ross, A. Cohen, and D. Kraemer. TREC 2004 genomics track overview. In *The Thirteenth Text Retrieval Conference (TREC 2004) NIST Special Publication SP 500-261*, 2004.

[12] L. Hirschman, A. Yeh, C. Blaschke, and A. Valencia. Overview of BioCreAtIvE: critical assessment of information extraction for biology. *BMC Bioinformatics*, 6(Suppl 1), 2005.

[13] M. Huang, X. Zhu, Y. Hao, D. G. Payan, K. Qu, and M. Li. Discovering patterns to extract protein–protein interactions from full texts. *Bioinformatics*, 20(18), 2005.

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

[15] J. Knowles and E. J. Hughes. Multiobjective optimization on a budget of 250 evaluations. In *Evolutionary Multi-Criterion Optimization (EMO 2005), LNCS 3410*, pages 176–190. Springer-Verlag, 2005.

[16] A. Koike, Y. Niwa, and T. Takagi. Automatic extraction of gene/protein biological functions from biomedical text. *Bioinformatics*, 21(7):1227–1236, April 2005.

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

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

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

[20] 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*, pages 80–81, 2001.

[21] M. F. Porter. An algorithm for suffix stripping. *Program*, 14(3):130–137, 1980.

[22] E. Riloff. Automatically constructing a dictionary for information extraction tasks. In *National Conference on Artificial Intelligence*, pages 811–816, 1993.

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

[24] A. Sehgal. Text mining: The search for novelty in text. Ph.D comprehensive examination report, Dept. of Computer Science, The University of Iowa, Apr. 2004.

[25] 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.