M. V. Tablan

Toward Portable

Information Extraction

Submitted for the degree of Doctor of Philosophy

2009
Toward Portable
Information Extraction

Mihai Valentin Tablan

Submitted for the degree of Doctor of Philosophy

Department of Computer Science
The University of Sheffield

December 2009
# Contents

Abstract v

Acknowledgements vii

1 Introduction 1

1.1 The Field: Information Extraction ......................... 1

1.2 Applications of Information Extraction ..................... 2

1.3 The Problem: IE Portability ............................... 4

1.4 Contribution and Structure of the Thesis ................... 5

2 Context and State-of-the-Art 7

2.1 Information Extraction – an Overview ..................... 7

2.2 A Closer Look at Information Extraction ..................... 11

2.2.1 Named Entity Recognition ............................... 14

2.2.2 Coreference Resolution ................................. 15

2.2.3 Template Element Production ........................... 17

2.2.4 Template Relation Production ........................... 18

2.2.5 Scenario Template Extraction ........................... 19

2.2.6 Domain Dependence .................................... 20

2.3 ACE - Toward Semantic Tagging of Entities .................. 21

2.4 Evaluation Methods ....................................... 24
## Contents

2.4.1 Cost-based Evaluation ........................................... 27
2.5 Applications of Information Extraction .......................... 27
2.6 Portability Work – Motivation and Related Approaches .... 32
    2.6.1 Adaptive Information Extraction ......................... 32
2.7 Summary ............................................................. 34

3 Portable Information Extraction: Two Approaches ............ 35
    3.1 Knowledge Engineering and Machine Learning for Information Extraction ........................................... 35
    3.2 Knowledge Engineering with JAPE ............................ 36
        3.2.1 JAPE – a Finite State Machine for Annotations ... 38
        3.2.2 Features of the JAPE Language ....................... 39
        3.2.3 Support for Ontologies in JAPE ....................... 49
        3.2.4 Initialisation of a JAPE Grammar .................... 53
        3.2.5 Execution of JAPE Grammars ......................... 60
    3.3 Machine Learning with OLLIE ................................. 67
        3.3.1 The OLLIE Application ................................. 68
        3.3.2 Collaborative and IE-supported Data Annotation ... 69
        3.3.3 Machine Learning Support ............................. 71
        3.3.4 OLLIE Experiments and Evaluation .................. 74
    3.4 Summary ............................................................. 79

4 Case Studies and Evaluation ........................................ 81
    4.1 Case Studies ...................................................... 81
        4.1.1 Case Study 1: Large Scale Semantic Annotation of Patent Documents ........................................... 81
        4.1.2 Case Study 2: Semantic Analysis of TV News Broadcasts ......................................................... 85
        4.1.3 Case Study 3: Ontology Authoring with Controlled Language Information Extraction .................. 86
A.4 Support for Evaluation ............................................ 131
A.5 Infrastructure for Multilinguality ............................ 134
   A.5.1 Unicode-compliant Language Resources ............ 134
   A.5.2 Processing Resources and Unicode ................ 136
   A.5.3 A Unicode-Aware Graphical Interface ............ 136
A.6 Processing Resources for Information Extraction ......... 139
   A.6.1 The Unicode Tokeniser .............................. 140
   A.6.2 The Gazetteer Look-up Component ................. 143
   A.6.3 The Sentence Splitter .............................. 144
   A.6.4 The Part of Speech Tagger ........................ 144
   A.6.5 The Named Entity Recogniser .................... 145
A.7 Working with Ontologies .................................... 146
   A.7.1 Data Model for Ontologies ....................... 147
   A.7.2 The Ontology Resource ............................ 149
   A.7.3 The Ontology Viewer .............................. 151

B Impact and Community Uptake ................................ 155

Bibliography .......................... 159
Abstract

In this thesis we present work which aims to reduce the effort required for creating new Information Extraction (IE) systems, or for adapting existing ones to new purposes. This makes IE techniques more cost-effective in areas where they are currently unsuitable, such as applications where the data volume is small, or where the information needs are changing frequently.

To support these dynamic needs of IE systems, we created JAPE – a formalism for specifying transduction rules over annotation graphs, and we implemented an execution environment for applying such rules. The features that distinguish JAPE from other pattern matching frameworks, such as regular expressions, include the use of a graph structure as input, and hierarchical matching based on ontologies.

In support of machine learning systems, we also developed OLLIE, an environment for computer-assisted collaborative annotation. It incorporates an algorithm for bi-directional translation between textual annotations and feature vectors that supports, in a generic fashion, the application of machine learning methods to any text annotation problem, including IE.
Acknowledgements

Whilst it is impossible to include all those that have helped in one way or another, thanks are due, in no particular order, to:

My research panel, consisting of Professors Yorick Wilks, Robert Gaizauskas, and Phil Green who have guided me with a steady hand throughout my postgraduate research period.

My friends and colleagues in the Sheffield NLP group, especially Hamish Cunningham, Kalina Bontcheva, and Genevieve Gorrell, who have willingly put up with early versions of this thesis, offered suggestions, and donated large chunks of their time helping with proofreading.

The community of GATE users who, through their bug reports, feature requests, and discussions on the mailing lists, have brought in a healthy dose of pragmatism.
Chapter 1

Introduction

The portability of Information Extraction systems has been addressed in previous work by learning rules or models in various ways. While learning has an important contribution to make, there are a number of other significant elements that influence portability, including the area of software development tools and the provision of infrastructure for language processing. This thesis reports work that addresses the wider issue of portability for extraction, and details results that contribute to this field.

1.1 The Field: Information Extraction

The move in recent years from paper-based documents towards electronic text along with the growth of the Internet and the accumulation of vast amounts of textual data in many commercial intranets has triggered the demand for a means of automatic access to the knowledge stored in electronic text.

Full automatic understanding of free text is beyond the reach of current natural language processing (NLP) applications and is most likely not achievable in the foreseeable future. However, research has shown that in many cases certain characteristics of documents and their content (such as genre, style, aspects of subject matter, layout, etc.) are sufficiently regular to allow some automatic analysis to be successfully performed.
The task of harvesting structured, quantifiable and unambiguous data from natural language text is known as Information Extraction (IE) [Cowie & Lehnert 96, Gaizauskas & Wilks 98, Appelt 99]. Various types of data may be extracted, starting from simple entity identification, through entity reference resolution and descriptive modifier attachment, to the identification of complex events and relations. The outputs of IE are records similar to database entries, which can be used for a wide range of operations such as data mining, automatic summarisation, or report generation.

IE has become a highly productive research area in the NLP field: a large number of R&D sites, both academic and commercial, being involved in technological advancements. One of the most important factors driving the rapid development of the field has been the Message Understanding Conference series (MUC), [ARPA 91, ARPA 92, ARPA 93, Sundheim 95, Sundheim 98], which played an important role in defining the area of research by establishing de facto standards that, in time, have become widely accepted by the community.

1.2 Applications of Information Extraction

IE has proved valuable in a number of application contexts, including:

**competitor intelligence:** e.g. tracking prices offered by competing on-line services [Fong et al. 01];

**summarisation:** e.g. extracting the most significant sections from long company reports referring to health and safety issues [Maynard et al. 02];

**question answering:** QA systems use IE techniques to extract facts from the text collections used as input [Moldovan et al. 07].

**word processing:** e.g. the ‘smart tags’ facility in recent versions of MS Word;

**e-science:** e.g. extracting plant taxonomy data from biological texts in order to compare and synthesise varying approaches [Wood et al. 03];
multimedia indexing: e.g. analysing football commentaries, news articles, and web pages relating to football matches, in order to index and semantically annotate videos of the matches [Saggion et al. 02];

digital libraries: e.g. marking-up eighteenth century criminal trial reports for humanities researchers [Bontcheva et al. 02] or generating semantic meta-data for audio-visual archives in order to improve access to the contents [Dowman et al. 05].

on-line collaboration: e.g. performing textual analysis by users at several geographically disjoint locations on a shared collection of texts [Tablan et al. 03].

The emergence of the Semantic Web initiative (SW) [Berners-Lee et al. 94, Berners-Lee 99] and the arrival of the first applications based on SW technologies promise a more user-friendly interaction between people and computers. These new capabilities rely on the presence of semantic metadata associated with the web documents and will allow the machines to access some of the meaning currently encoded only in textual form, and thus to perform useful actions based on that semantics.

However, the overall majority of existing web pages do not have associated semantic metadata and cannot benefit from these new developments. In order to enable the use of SW tools, much of the textual information currently present on the Internet needs to be enhanced by the addition of semantic annotation, and this is one of the main hurdles that hinders the spread of SW.

We believe that one possible answer to this problem is to use Information Extraction to automatically or semi-automatically create the necessary semantic annotation directly from the textual content of existing documents. This is already becoming apparent in current SW research where NLP techniques, and especially IE, are being applied to:

ontology creation: using information present in text to derive a domain ontology or to extend an existing one with data relating to a new domain. One example is creating an ontology describing semantic
web services by analysing the textual documentation associated with software [Sabou 04].

**ontology population:** deals with enriching the data in an ontology by adding new instances of the classes already present. The KIM platform [Popov et al. 04] performs such an action; it starts with a ontology containing known entities and then it enriches it with information extracted from web pages of news web sites that are crawled regularly.

**semantic annotation:** refers to the generation of semantic meta-data for existing textual resources based on Information Extraction techniques, e.g. AeroDAML [Kogut & Holmes 01] is a tool that demonstrates this use of IE techniques.

### 1.3 The Problem: IE Portability

IE systems are labour-intensive to develop, requiring the expertise of highly skilled personnel. Currently, deployments of IE technology are only cost-effective when used for large volumes of textual data and for static information needs.

The changing nature of the information need and the diversity of application areas, for instance when used in the Semantic Web context, creates a demand for IE that can be quickly and easily ported to new tasks. This is the challenge addressed by this thesis.

The need for IE systems that can be easily adapted from one domain to another was identified early in the life of the research field as, illustrated by the work in *adaptive information extraction* starting with the Alembic Workbench [Aberdeen et al. 95] in 1995 and, more recently, the Melita system [Ciravegna et al. 03], or the work in wrapper induction [Freitag & Kushmerick 00]. The idea behind these approaches was to use various kinds of machine learning algorithms to allow IE systems to be easily targeted to new problems. The effort required to redesign a new system was replaced with that of generating a new batch of training data and applying the learning algorithm. Given that the work for generating training data comprises mainly manual annotation of text, which requires
less skilled personnel, this brings down the overall cost for re-targeting a system to a new domain.

While the use of machine learning helps (and is supported by the work reported here), there are a variety of other means for reducing the effort required in creating a new IE system. They are all encompassed here by the term **portability** which reflects the diverse challenges faced when attempting to adjust to changes in the definition of the problem that is being addressed and the text being processed.

This thesis will address multiple dimensions of IE portability, including:

**task:** different tasks involve different combinations of IE data types of varying complexity;

**genre and style:** the language involved varies from formal and precise (e.g. news articles) through informal and badly spelled (e.g. emails);

**domain:** the subject matter of the texts to be analysed;

**scale:** IE applications vary from the small-scale to the size of the web itself, and processing requirements vary from overnight to near real-time;

**end-user type:** some users may have a sophisticated knowledge of computing and/or linguistics, whereas others may have almost none;

**language:** the same IE task may need to be applied to inputs in different languages;

1.4 Contribution and Structure of the Thesis

The chapter following this sets the scene by presenting some background on Information Extraction. It includes a short history of the field, describes the main sub-areas, and the current state of the art.

The main contribution of this thesis is presented Chapter 3. It consists of two approaches for improving the portability of Information Extraction systems.
Chapter I: Introduction

The first part of the chapter, Section 3.2, introduces the provisions we developed for IE systems based on manually-created rules. It describes a representation language for transduction rules as well as an execution environment for applying such rules. Together they provide a way of performing Information Extraction tasks by mechanically processing textual documents. Because the rules can be grouped together in phases that are applied one after the other in a cascade, a high level of modularity is afforded. This leads to localisation of changes: many variations in the problem definition can be accommodated by modifications of individual phases while the rest of the system is left unchanged.

The second part of Chapter 3, mainly Section 3.3, reports work we did to support the use of machine learning methods. It presents a system for on-line collaborative annotation, that uses Machine Learning techniques to create models that can perform Information extraction tasks. It also includes the implementation of a general purpose library for extracting features from textual data that can then be used to train a classification model.

These two contributions, together with the wide-ranging infrastructural support described in Appendix A, lead to a reduction in the effort required for dealing with changes along the portability dimensions described in the previous section.

Chapter 4 is dedicated to presenting a set of four case studies that are then analysed to extract data for quantitative evaluation. Some additional non-quantitative evaluation elements are included in Appendix B, which talks about the community impact of this work.

Finally, Chapter 5 concludes the thesis and discusses plans for future work.
Chapter 2

Context and State-of-the-Art

2.1 Information Extraction – an Overview

The first algorithms for IE were developed in universities at the beginning of the 1980s and were usually based on keywords and skilfully designed sets of rules. For example, the FRUMP system [DeJong 82], developed in 1982, was using manually created scripts to process news stories. An even earlier example is the work in the early 70s of Naomi Sager of the Linguistic String Project at New York University, which dealt with the formatting of natural language information, see for example [Sager 78]. Its aim was to automatically fill-in database records by analysing natural language text, a process we now recognise as Information Extraction.

These early systems were relatively simple and would be seen as naïve today. They made little or no use of linguistic features and had very little flexibility with regard to the style, genre or domain of the input texts. They were highly targeted systems and their performance was achieved by tedious fine-tuning of the set of heuristics employed.

While the systems mentioned above are some of the first examples, a notable role in crystallising the range of efforts in the emerging field of Information Extraction was played by the MUC\(^1\) series of conferences. They were organised between 1987 and 1998 by NRAD, the RDT&E division of the

\(^1\)MUC stands for Message Understanding Conference
Chapter II: Context and State-of-the-Art

Naval Command, Control and Ocean Surveillance Center (formerly NOSC, the Naval Ocean Systems Center) with the support of DARPA, the Defense Advanced Research Projects Agency. The conferences were a medium for reporting the work and results obtained in associated competitions for IE systems.

“For each MUC, participating groups have been given sample messages and instructions on the type of information to be extracted, and have developed a system to process such messages. Then, shortly before the conference, participants are given a set of test messages to be run through their system (without making any changes to the system); the output of each participant’s system is then evaluated against a manually-prepared answer key.”

[Grishman & Sundheim 96a]

Funding for US laboratories was made available through the TIPSTER programme which attracted a high level of interest from American research institutions.

The MUC competitions were particularly relevant because they gave birth to an IE scientific community and because they coagulated all the random efforts in the field and gave them a common direction. It was in the context of MUC that the domain of IE was shaped by the creation of the first clear definition of the problem and the introduction of community-accepted evaluation metrics. Because almost all parties interested in the development of IE took part in MUC, the decisions taken in that forum were accepted as a de facto standard which helped the advancement of research by providing a level playing field.

A brief overview of the main developments during MUC conferences includes:

---

2The TIPSTER Text Program was a DARPA led government effort to advance the state of the art in text processing technologies through the cooperation of researchers and developers in government, industry and academia. The resulting capabilities were deployed within the intelligence community to provide analysts with improved operational tools. The programme formally ended in the autumn of 1998.
### MUC-1 (1987)
Was mostly exploratory – each group was using its own representation model and there was no formal evaluation at the end.

### MUC-2 (1989)
Defined the task as a template filling one (10 slots) and introduced the evaluation measures of recall and precision. Both MUC-1 and MUC-2 used military messages about naval sightings and engagements as input texts.

### MUC-3 (1991) and MUC-4 (1992)
The texts used were reports of terrorist events in Central and South America and the templates had 18/24 slots.

### MUC-5 (1993)
Was the first to be conducted as part of the TIPSTER programme and represented a leap in the complexity of the task: two types of text were used – international joint ventures and electronic circuit fabrication, in two languages – English and Japanese. The joint venture task required 11 templates with a total of 47 slots for the output.

### MUC-6 (1995)
Was the first to identify the need for more portability for IE systems. Its stated goals were:
- to identify components that are task-independent, can be performed automatically with reasonable accuracy and have practical uses;
- to focus on portability in the task, defined as the ability to rapidly re-target a system to extract information about a different class of events;
- to encourage deeper understanding by requiring the systems to perform co-reference resolution, word sense disambiguation and predicate-argument structure extraction.

This resulted in breaking the generic IE task into a set of sub-tasks: Named Entity, Coreference, Template Element and Scenario Template; each participating site could choose to take part in the competition for all or only a subset of the tasks [Grishman & Sundheim 96b].
Chapter II: Context and State-of-the-Art

MUC-7 (1998)

The task definition was similar to that of MUC-6, the only addition being the Template Relation sub-task.

At the end of the MUC conferences, IE emerged as a well-defined problem, with a large number of laboratories actively developing systems and working on the advancement of the state-of-the-art.

The commonly accepted definition of IE is:

An automated process which takes unseen texts as input and produces fixed-format, unambiguous data as output.

This data may be used directly for display to users, or may be stored in a database or spreadsheet for later analysis, or may be used for indexing purposes in Information Retrieval (IR) applications such as Internet search engines.

Information Retrieval is a better known technology because it has come to public attention through the popularity of web indexing engines such as Google. It is instructive to compare IE and IR: whereas IR finds relevant texts and returns them as output, the typical IE application analyses texts and only returns the precise piece of information that the user is seeking.

It is hard to compare the usefulness of IR and IE; they are both technologies that assist people with solving information-seeking problems. While IR directs the attention of the user to the relevant texts from a possibly very large collection, IE further focuses the search towards the relevant pieces of information inside each individual text. IR and IE are complementary, and can be used together to automate as much as possible the task of information finding.

One outcome from the MUC effort was the realisation that IE can successfully be performed using quite shallow methods, for instance those based on regular expressions and finite state machines. One interesting example is provided by the systems entered into the competition by the Artificial
Intelligence Center at SRI International\(^3\). For MUC-3, SRI developed a system called TACITUS [Hobbs 91] which was based on deep analysis of the input text. Although that provided good quality results, the complexity of the processing made it very slow and hindered development efforts. As a consequence, for the next MUC evaluations SRI decided to develop a brand new system, called FASTUS [Hobbs et al. 96] based largely on finite state processing which provided similar levels of performance while increasing the execution speed by several orders of magnitude. A result of this experience and other similar ones, is that it has become widely accepted that shallow methods are better suited for addressing IE requirements, not least because one motivation behind IE research is the need to process vast quantities of text and, as such, the efficiency of the systems is important. By the end of the MUC conference series, most IE systems were employing shallow, finite-state like formalisms, see e.g. [Cowie et al. 93, Gaizauskas et al. 95, Grishman 95, Day et al. 98].

The next section looks at the various sub-tasks of modern IE systems, their performance levels and highlights the issue of *domain dependence*.

### 2.2 A Closer Look at Information Extraction

There are five types of IE (or IE *tasks*) currently under R&D (as defined by the leading forum for this research, the Message Understanding Conferences [Grishman & Sundheim 96a, SAIC 98]):

<table>
<thead>
<tr>
<th>Named Entity recognition (NE)</th>
<th>Finds and classifies names, places, etc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coreference resolution (CO)</td>
<td>Identifies identity relations between enti-</td>
</tr>
<tr>
<td></td>
<td>ties in texts.</td>
</tr>
<tr>
<td>Template Element construction (TE)</td>
<td>Adds descriptive information to NE re-</td>
</tr>
<tr>
<td></td>
<td>sults (using CO).</td>
</tr>
<tr>
<td>Template Relation construction (TR)</td>
<td>Finds relations between TE entities.</td>
</tr>
</tbody>
</table>

\(^3\)http://www.ai.sri.com
Chapter II: Context and State-of-the-Art

| Scenario Template production (ST) | Fits TE and TR results into specified event scenarios. |

In simpler terms: NE is about finding entities; CO about which entities and references (such as pronouns) refer to the same thing; TE about what attributes entities have; TR about what relationships between entities there are; and ST about events that the entities participate in. Consider these sentences:

Deutsche Telekom yesterday delayed the demerger of its T-Mobile wireless business, which owns UK network One2One, until next year owing to the volatility of the stock markets.

NE discovers that the entities present are the *Deutsche Telekom*, *yesterday*, *T-Mobile*, *UK*, *One2One*, *next year*. CO discovers that *its* in the first sentence refers to *Deutsche Telekom*. TE finds that *T-Mobile* is involved in the *wireless* sector, TR that *T-Mobile* is part of *Deutsche Telekom*, that *One2One* is based in the *UK* and that *T-Mobile* owns *One2One*. Finally, ST recognises an event (the delaying of a demerger) in which the various entities are involved.

The various types of IE tasks provide progressively higher-level information about texts. They are described in more detail below. A typical input text for an IE system is shown in Figure 2.1.

Each of the five sub-tasks of IE has been the subject of rigorous performance evaluation in MUC-7 (1998) [SAIC 98] and other MUCs, so it is possible to say quite precisely how well the current level of technology performs. Below we will quote percentage figures quantifying performance levels, which should be interpreted as a combined measure of Precision and Recall (or ‘F-Measure’, see the section 2.4 for details on these evaluation metrics). Two caveats should be noted: most of the evaluation has been on English (with some on Japanese, Chinese and Spanish); and some applications of the technology may be easier or more difficult in different languages.

The performance of each IE task, and the ease with which it can be developed, is to varying degrees dependent on:
Toward Portable Information Extraction

**Figure 2.1: An example text**

**Text type:** the kinds of texts we are working with, for example: Wall Street Journal articles, email messages, HTML documents from the Web, or the output of a speech recogniser.

**Domain:** the broad subject matter of those texts, for example: financial news, requests for technical support, or tourist information, and the style in which they are written, e.g. informal, formal.

**Scenario:** the particular event types that the IE user is interested in, for example: mergers between companies, problems experienced with a particular software package, or descriptions of how to locate parts of a city.

A particular IE application might be configured to process financial news articles from a particular news provider and find information about mergers
between companies and various other scenarios. The performance of the application would be predictable for only this set of parameters. If it was later required to extract opinions about software packages expressed by users of an on-line forum, performance levels would no longer be predictable. Tailoring an IE system to new requirements is a task that varies in complexity dependent on the degree of variation in the three factors listed above.

2.2.1 Named Entity Recognition

The simplest and most reliable IE technology is Named Entity recognition (NE). NE systems identify all the names of people, places, organisations, dates, amounts of money, etc. So, for example, if we run the text in figure 2.1 through an NE recogniser, the result is as in Figure 2.2.

![Figure 2.2: Named Entity recognition](image)

NE recognition can be performed at up to around 95% accuracy. Given
that human annotators do not perform to the 100% level (measured in MUC by inter-annotator comparisons), NE recognition can now be said to function at human performance levels, and applications of the technology are increasing rapidly as a result. It may be surprising that humans cannot reach 100% accuracy. This can be explained by the fact that the rules as to what constitutes an entity, as presented in annotation manuals, are very rigid. People, on the other hand, have a tendency to generalise from examples, to create implicit rules which may not always coincide with the official ones. Another factor is that the activity of manually annotating text is very repetitive and tedious. This can lead to loss of concentration on the part of the human annotators, which explains why some errors are being made.

An evaluation of NE for Spanish, Japanese and Chinese ([Merchant et al. 96]) produced the following scores:

<table>
<thead>
<tr>
<th>Language</th>
<th>Best System</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spanish</td>
<td>93.04 %</td>
</tr>
<tr>
<td>Japanese</td>
<td>92.12 %</td>
</tr>
<tr>
<td>Chinese</td>
<td>84.51 %</td>
</tr>
</tbody>
</table>

The process of NE recognition is not affected by the text type, as long as the quality of the text is not degraded as in the case of speech recognisers’ output.

### 2.2.2 Coreference Resolution

Coreference resolution (CO) involves identifying identity relations between entities in texts. Besides entities identified by NE recognition, this may also include anaphoric references to those entities. For example, in the text presented in Figure 2.3, a pronominal coreferencer would also recognise that the its pronoun refers back to Deutsche Telekom.

This process is less relevant to users than to other IE tasks, i.e. whereas the other tasks produce output that is of obvious utility for the application user, this task is more relevant to the needs of the application developer. For text browsing purposes we might use CO to highlight all occurrences of
the same object or provide hypertext links between them. CO technology might also be used to make links between documents, though this was not part of the MUC programme. The main significance of this task, however, is as a building block for TE and ST (see below). CO enables the association of descriptive information scattered across texts with the entities to which it refers. Figure 2.3 shows the results of running an entity coreferencer for the example text from Figures 2.1 and 2.2.

Figure 2.3: Coreference resolution

CO resolution is an imprecise process when applied to the problem of anaphoric reference. CO results vary widely: depending on the domain, perhaps only 50-60% may be relied upon. These scores are low and suggest that CO results cannot be used reliably. However, when applied in an IE context, CO resolution between named entities is a simpler problem as it reduces in many cases to proper-noun coreference identification (same object, different spelling or compounding, e.g. ‘IBM’, ‘IBM Europe’, ‘International Business Machines Ltd.’, …). In this case, CO resolution can be done with
higher levels of accuracy and can be a useful step.

CO systems are not affected by text type but they are weakly domain dependent.

2.2.3 Template Element Production

The TE task builds on NE recognition and coreference resolution. Its role is to associate descriptive information with the entities. For example, from the text in Figure 2.1, the system finds out that One2One is based in the UK and that Deutsche Telekom is partly state-owned. Some of the template elements for the text in the Figure 2.1 are given in Figure 2.4.

```
Date-0001:
  Value: 01/08/2001
  Alias: Wednesday August 1, 2001
Organisation-0001:
  Name: Deutsche Telekom
  Type: company
  Ownership: public/private
Date-0002:
  Value: 31/07/2001
  Alias: yesterday
Organisation-0002:
  Name: T-Mobile
  Type: company
  Ownership: public/private
Location-0001:
  Name: UK
  Type: country
Organisation-0003:
  Name: One2One
  Type: company
  Location: UK
```

Figure 2.4: Template Elements

The format presented here is somewhat arbitrary, inspired from the one developed for the MUC competitions. It is difficult to read; the main point to note is that it is essentially a database record, and could just as well be
formatted for SQL store operations, or reading into a spreadsheet, or (with some extra processing) for multilingual presentation.

The best MUC-7 system scored around 80% for TE, while humans achieved 93%. MUC-6 was the first MUC to evaluate TE separately of the final ST task. This reflects that, although its primary use is as an aid for the downstream modules, it is a process which could be used on its own. An example of applying TE is the creation of entity profiles shown by the KIM tool [Popov et al. 04].

The production of TEs is domain dependent, as the types of information that are relevant depend on the types of entities that are important to the application domain. For example, relevant information about an organisation includes whether it is private or public, if it is for profit or a charity. By contrast, useful data about academic papers might be whether they were published in the proceedings of a workshop, a conference, or in a journal. These different types of information require different methods for extraction.

### 2.2.4 Template Relation Production

Before MUC-7, relations between entities were part of the scenario-specific template outputs of IE evaluations. In order to capture more widely useful relations, MUC-7 introduced the **TR** task.

The template relation task requires the identification of a small number of possible relations between the template elements identified in the template element task. This might be, for example, an employee relationship between a person and a company, a family relationship between two persons, or a subsidiary relationship between two companies. Extraction of relations among entities is a central feature of almost any information extraction task, although the possibilities in real-world extraction tasks are endless. [Appelt 99]

Coming back to the example text in Figure 2.1, and assuming that the domain of the application is financial news, the result of TR would be
recognising the fact that T-Mobile is part of Deutsche Telekom and that T-Mobile owns One2One.

In MUC-7 the best TR scores were around 75%.

The line between TE and TR is somewhat indistinct as both identify information relating to entities found by NE. What separates them is the domain of the application: TR finds relations between entities, with both the relation and entity types being relevant to the application domain. TE finds additional information about entities, which may involve other entities (like the fact the One2One is based in the UK) but this data is mainly used to enrich the description of the entity.

Like TE, TR is a domain dependent task – the types of entities involved in relations as well as the relations themselves depend on the application domain, and the extraction methods can vary widely.

### 2.2.5 Scenario Template Extraction

Scenario templates (STs) are the prototypical outputs of IE systems, being the original task for which the term was used. They tie together TE entities and TR relations into event descriptions. An example, describing a management transition event, is given in Figure 2.5.

![Figure 2.5: Scenario Template](image)

ST is a difficult IE task; the best MUC systems score around 60%. The human score can be as low as around 80%, which illustrates the complexity involved. These figures should be taken into account when considering appropriate applications of ST technology.
It is noteworthy that in MUC-6 and MUC-7 the developers were given the specifications for the ST task only 1 month before the systems were scored. This was because it was considered that an IE system that required very lengthy revision to cope with new scenarios was of less worth than one that could meet new specifications relatively rapidly. As a result of this, the scores for ST in MUC-6/7 were probably slightly lower than they might have been with a longer development period. Experience from previous MUCs suggests, however, that current technology at that time had difficulty attaining scores much above 60% accuracy for this task.

The ST task is both domain dependent and, by definition, tied to the scenarios of interest to the users.

### 2.2.6 Domain Dependence

The influence of domain dependence on the effectiveness of NLP tools such as IE systems is an issue that is all too frequently overlooked. IE systems mostly extract fixed information from documents in a particular language and domain. For the technology to be suitable for real-world applications, IE systems need to be easily customisable to new domains [Karkaletsis et al. 99].

Due in no small part to the MUC competitions (e.g. [Sundheim 95, Sundheim 98]), work on IE has largely focused on narrow sub-domains. For example MUC 3 and MUC 4 focused on newswires about terrorist attacks, while MUC 7 was concerned with reports on air vehicle launches. Some work has been carried out on adapting existing systems to new domains, but there has been little success in making a single system robust enough to deal with different domains. The adaptation of existing systems to new domains requires a substantial amount of knowledge, and its acquisition and application are non-trivial tasks. For IE systems, the complexity of the domain may be particularly influential [Bagga 98].

An independent (though related) issue concerns the adaptation of existing systems to different text genres. By this we mean not just changes in domain, but different media (e.g. email, spoken text, written text, web pages), text type (e.g. reports, letters, books), and structure (e.g. layout). The genre of a text may be influenced by a number of factors, such as author, intended
audience and degree of formality. For example, less formal texts may not follow standard capitalisation, punctuation or even spelling rules, all of which can be problematic for the intricate mechanisms of IE systems.

The problem of domain dependence is also related to the issues of algorithmic reuse. It seems to be particularly true of IE systems that they are tailor-made for specific domains and applications, with the result that not only are they hard to adapt for new tasks, but that it is difficult to extricate potentially reusable components or sub-components which are buried deep in the architecture. For example, there may not be a distinction between foreground information (which is domain-, and application-dependent), and background information (which can be reused as it stands); and consequently, between the tools needed to access and manipulate these two types of information.

2.3 ACE - Toward Semantic Tagging of Entities

The success of the MUC series, both in terms of the number of participating sites and the level of advance in the field of IE, has encouraged the organisation of a new competition: the ACE conferences [ACE04a, ACE04b, ACE04c]. This is a successor of MUC, but definitions of tasks as well as the basic concepts (such as types of Named Entities) are slightly different.

While MUC NE task tags segments of text whenever that text represents the name of an entity, in ACE, these names are viewed as merely mentions of the entities that exist externally to the text. The main task is to detect (or infer) the existence of the entities themselves. This essentially groups together the NE and CO tasks from MUC; a coreference error might mean recognising two mentions of the same entity as two independent entities, thus identifying a spurious entity for which the system will be penalised.

ACE defines the following tasks:

- *Entity Detection and Tracking (EDT)* - the task of identifying named entities in texts. This task roughly corresponds to the NE and CO tasks in MUC. The types of entities considered are different from MUC:

\[4\text{ACE stands for Advanced Content Extraction.}\]
– Person (incl. pronouns and nominal mentions)
– Organisation (incl. Government, Commercial, Educational, Non-profit, Other)
– Location (incl. Address, Boundary, Celestial, Water-Body, Region-Natural, Region-City-or-Town, Region-Provincial, Region-National, Region-Global, Other)
– Facility (incl. Plant, Building, Bounded-Area, Conduit, Path, Barrier, Other)
– Weapon
– Vehicle (incl. Air, Land, Water, Other)
– Geo-Political Entity, GPE (incl. Continent, Nation, State-or-Province, County-or-Prefecture, City-or-Town, Other)

It should be noted that the named entity taxonomy employed by ACE is more finely grained than MUC.

• Relation Detection and Characterisation (RDC) - the tracking of identifying relations between entities (roughly corresponding to the TE and TR). Relation types include:
  – physical relations (incl. Located, Near and Part-Whole)
  – social/personal relations (incl. Business, Family and Other)
  – employment or membership relations
  – ownership relations
  – affiliation-type (ethnicity, citizenship, etc)

• Event Detection and Characterisation (EDC) - identifying atomic events the entities participate in (roughly corresponding to the ST task in MUC). Event types include:
  – Interaction
  – Movement
  – Transfer
  – Creation
Entity detection output is in the form of a (unique) ID for the entity, a set of entity attributes, and information about the way in which the entity is mentioned: document ID, mention type (one of Name, Nominal or Pronominal), mention head and mention extent. An example entity output is shown in Figure 2.6.

```xml
<entity ID="ft-airlines-27-jul-2001-2"
   GENERIC="FALSE"
   entity_type = "ORGANIZATION">
   <entity_mention ID="M003"
       TYPE = "NAME"
       string = "National Air Traffic Services">
   </entity_mention>
   <entity_mention ID="M004"
       TYPE = "NAME"
       string = "NATS">
   </entity_mention>
   <entity_mention ID="M005"
       TYPE = "PRO"
       string = "its">
   </entity_mention>
   <entity_mention ID="M006"
       TYPE = "NAME"
       string = "Nats">
   </entity_mention>
</entity>
```

Figure 2.6: Sample entity output

The complexity of the ACE tasks is much higher than that of the equivalent MUC tasks because of the following factors:

- Deeper analysis involved: the MUC NE task performs linguistic analysis of the text while the corresponding ACE EDT task has to deal with semantic analysis as well. A named entity in ACE is represented by its mentions (named, pronominal or nominal) and the system has to infer the entities and the roles they play.
Chapter II: Context and State-of-the-Art

- Open domain: the ACE data covers multiple domains such as sport, politics, business, culture and religion.
- Conflating MUC tasks: the MUC CO task (which had quite low performance) is part of the EDT task and thus a poor coreference resolution handling has negative impact on the entity detection scores.

The differences between the task definitions of MUC and ACE underline a trend to push forward the IE techniques in order to extract deeper semantics from text. This is further encouraged by the development of technologies emerging from Semantic Web research that require semantic meta-data as input. A new-generation IE is the ideal candidate for automatically extracting semantics from legacy text, thus enabling SW tools to overcome the barrier caused by the initial lack of semantic meta-data.

2.4 Evaluation Methods

When you can measure what you are speaking about, and express it in numbers, you know something about it; but when you cannot measure it, when you cannot express it in numbers, your knowledge is of a meager and unsatisfactory kind: it may be the beginning of knowledge, but you have scarcely in your thoughts advanced to the stage of science.

(Lord Kelvin, 1883)

In order to measure progress, suitable evaluation methods are necessary. IE research has benefited from its inception from well-defined rigorous evaluation methods. This section describes some of the most popular metrics used to quantify performance of IE systems.

As discussed in Section 2.1, much of the research in IE in the last decade has been connected with the MUC competitions, and so it is unsurprising that the MUC evaluation metrics of precision, recall and F-measure [Chinchor 92] also tend to be used, sometimes with slight variations. These metrics have a very long-standing tradition in the field of IR [van Rijsbergen 79] (see also [Manning & Schütze 99, Frakes & Baeza-Yates 92]).
Precision measures the number of correctly identified items as a percentage of the number of items identified. In other words, it measures how many of the items that the system identified were actually correct, regardless of whether it also failed to retrieve correct items. The higher the precision, the better the system is at ensuring that what is identified is correct.

Error rate is the inverse of precision, and measures the number of incorrectly identified items as a percentage of the items identified. It is sometimes used as an alternative to precision.

Recall measures the number of correctly identified items as a percentage of the total number of correct items. In other words, it measures how many of the items that should have been identified actually were identified, regardless of how many spurious identifications were made. The higher the recall rate, the better the system is at not missing correct items.

Clearly, there must be a trade-off between precision and recall, for a system can easily be made to achieve 100% precision by identifying nothing (and so making no mistakes in what it identifies), or 100% recall by identifying everything (and so not missing anything). The F-measure [van Rijsbergen 79] is often used in conjunction with Precision and Recall, as a weighted harmonic mean of the two.

False positives is a useful metric when dealing with a wide variety of text types, because it is not dependent on relative document richness\(^5\) in the same way that precision is.

Document richness is relevant in this context because if a particular document type has a significantly different number of any type of entity, the results for that entity type can become skewed. Compare the impact on precision of one error where the total number of correct entities is 1, and one error where the total is 100. Assuming the document length is the same, then the false positive score for each text, on the other hand, should be identical.

Common metrics for evaluation of IE systems are defined as follows:

\(^5\)By this we mean the relative number of entities of each type to be found in a set of documents.
(2.1) \[
\text{Precision} = \frac{\text{Correct}}{\text{Correct} + \text{Spurious}}
\]

(2.2) \[
\text{Recall} = \frac{\text{Correct}}{\text{Correct} + \text{Missing}}
\]

(2.3) \[
F - \text{measure} = \frac{(\beta^2 + 1)P \cdot R}{(\beta^2 R) + P}
\]

where \(\beta\) reflects the weighting of \(P\) vs. \(R\), a \(\beta\) value of 1 gives both \(P\) and \(R\) equal weights while a value of 0 effectively eliminates \(P\) from the equation.

(2.4) \[
\text{FalsePositive} = \frac{\text{Spurious}}{c}
\]

where \(c\) is some constant independent from document richness, e.g. the number of tokens or sentences in the document.

A variation of these measures is used when the concept of partially correct response is introduced. An annotation is said to be partially correct if the entity type is correct but the text span identified is only partially covering the actual entity mention. Partially correct responses are usually allocated a half weight. In this case the formulae become:

(2.5) \[
\text{Precision} = \frac{\text{Correct} + \frac{1}{2}\text{Partial}}{\text{Correct} + \text{Spurious} + \text{Partial}}
\]

(2.6) \[
\text{Recall} = \frac{\text{Correct} + \frac{1}{2}\text{Partial}}{\text{Correct} + \text{Missing} + \text{Partial}}
\]
2.4.1 Cost-based Evaluation

Beside F-measure, another method of getting a single numeric value to measure performance is the cost-based metric. This appears to be becoming a favourite with the DARPA competitions, such as ACE. The model stems from the field of economics, where the standard model, “Time Saved × Salary” uses the direct salary cost to an organisation as a measure of the value.

One of the main advantages of this type of evaluation is that it enables the evaluation to be adapted depending on the user’s requirements. A cost-based model characterises the performance in terms of the cost of the errors (or the value of the correct things). For any application, the relevant cost model is applied, and expected prior target statistics are defined.

For a cost-based error model, a cost would typically be associated with each type of errors (such as a false negative and a false positive, and with each category of result (e.g. recognising Person might be more important than recognising Date correctly). Expected costs of error would typically be calculated based on probability (using a test corpus). This makes the assumption that a suitable test corpus is available, which has the same rate of entity occurrence (or is similar in content) to the evaluation corpus. If necessary, the final score can be normalised to produce a figure between 0 and 1, where 1 is a perfect score. The main advantage of the cost model is it can be finely tuned to different applications by changing the application model parameters.

2.5 Applications of Information Extraction

As already outlined, the various IE tasks have different complexity levels and state of the art systems can attain varying degrees of accuracy. While some, like named entity recognition, are close to human performance, others are less successful, and their usefulness outside of research laboratories might be only marginal. Despite its shortcomings, IE technology is becoming increasingly popular in real-world applications for two main reasons. Firstly, sustained research efforts by both academic and industrial sites keeps pushing the
quality of results upwards, bringing more and more IE applications in the range of acceptable accuracy. The rapid development of the world wide web has also moved the goalposts: there is now so much information available on the net that even a system that does not manage to extract everything it could, can still be useful. The definition of what constitutes IE has also evolved since MUC and today there are many applications that use IE techniques although they would not necessarily fit within the scheme defined by MUC. They are, however, all performing the same task – that of extracting structured information out of unstructured data. The second, and more important, factor driving the uptake of IE is the need for such a tool. Information is now a very valuable asset and the quantities of textual data are increasing at an accelerating pace. It is no longer feasible to manage all the information need through the work of human analysts. The information needs of today’s society and the impossibility of addressing them through traditional methods, has led to IE technology being adopted even though it is not perfect.

A comprehensive review of IE research is presented in [Sarawagi 08]. It includes a simple taxonomy of IE applications, which we summarise in the table below.

<table>
<thead>
<tr>
<th>Enterprise Applications</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>News Tracking</td>
<td>Identifying items of interest in news feeds. This is the most typical application of IE and featured prominently in the MUC and ACE competitions.</td>
</tr>
<tr>
<td>Customer Care</td>
<td>Many corporate entities accumulate large amounts of unstructured information from their interaction with customers. This can include emails, forum postings, customer complaints, support requests, etc. Benefits can be obtained from integrating these data sources with the enterprise’s own structured databases or ontologies. This kind of problem setting has given rise to new extraction requirements, such as identifying product names, extracting customer names and addresses from invoices, automatically linking customer emails with order numbers.</td>
</tr>
<tr>
<td>Data Cleaning</td>
<td>In data warehousing applications the issue of data normalisation is of great importance. The same type of information can appear in many different forms, for example addresses expressed as plain text show a great degree of variability. IE can help with mapping these textual descriptions to structured database entries.</td>
</tr>
<tr>
<td>Classified Ads</td>
<td>IE techniques are now being applied to expose the implicit structure present in lists of classified ads, restaurants, film showing times, etc. The output of such a process can then be used to support querying.</td>
</tr>
</tbody>
</table>

**Personal Information Management**

PIM systems seek to organise personal data such as contacts lists, event calendars, emails, documents, multimedia files. IE can help by automatically identifying relevant information, such as the author of a document, or the time and place of an event mentioned in an email. This then supports the creation of hyperlinks, or the automatic population of calendars.

**Scientific Applications**

IE is applied to many areas of scientific research, the most prominent of which is bio-informatics. A typical example is the automatic recognition of biological entities, such as genes, or proteins from large publication databases, like Pubmed.

**Web Oriented Applications**

| Citation Databases | Several citation databases have been created, examples of which are *Citeseer* and *Google Scholar*. Information Extraction plays an important role in building them and it used on multiple levels: extracting individual publication records from lists such as conference web sites, personal web pages, bibliographical sections; breaking-up a publication record into individual fields; identifying records that refer, in different surface forms, to the same publication. |
| **Opinion Databases** | There are many sources on the Internet that contain unmoderated opinions about a range of topics including products, books, artists, films, etc. These are usually plain text sources, such as blogs, forums, newsgroup posts, review websites. IE can be used to structure this information and compile higher-level views that present e.g. the prevalent polarity of opinion (positive or negative) relating to any particular item. |
| **Community Websites** | Another example of creating structured databases from webpages are community websites such as DBLife⁶ or Rexa⁷ which track information about researchers, conferences, talks, and events relevant to a particular community. The creation of these databases relies on extracting information from academic webpages, or conference websites. |
| **Comparison Shopping** | Websites are now being created that support shoppers by comparing similar offerings from multiple suppliers. Their construction involves crawling merchant websites and extracting relevant information, such as product names, pricing, conditions of sale. |
| **Ad Placement on Webpages** | The return obtained from placing online adverts can be improved by optimising their placement on appropriate web pages. This type of functionality can be helped by using IE techniques to identify e.g. product names, mentions of competitors, or opinions referring to them. |
| **Structured Web Searches** | IE can be used to support web searches that go beyond the simple keyword-based paradigm, such as searching for relationships between entities. This can include looking for people that are employed by a given organisation, companies that have been acquired by others, organisations located in certain geographical areas. |

---

⁶http://dblife.cs.wisc.edu/
⁷http://rexa.info
The list of applications for IE presented above is not exhaustive. The remainder of this section contains some other examples that illustrate how IE applications can mediate between the text and the structured information needs of various types of user.

**Multimedia Archives**

IE methods have been used to automatically derive semantic meta-data for audio-visual archives. Large broadcasting organisations manually annotate the items in their archives in order to enable efficient access and retrieval of the materials. Due to the large volumes of new programs and to the large costs associated with manual annotation, only a fraction of their output is annotated with semantic data while the rest only has basic annotation like the programme name. The system described in [Dowman et al. 05] uses IE and Semantic Web techniques to derive such meta-data automatically.

**Question Answering Systems**

Question Answering systems sometimes rely on IE to build up a database of known entities and facts, which are then used when answering questions. For example, a consistently high-scoring system in the TREC QA tracks (named *PowerAnswer*, [Moldovan et al. 07]) uses a Named Entity Recogniser to identify names of people, organisations, dates, places, and others in the input texts.

**Dialogue Systems**

Less commonly, some Dialogue systems also employ IE techniques. The BirdQuest system [Jönsson et al. 04] uses Information Extraction to populate a database of facts about birds starting from a text book. This database is then used, in conjunction with other structured information sources, as an input to the dialogue management system.

**Semantic Web**

The Semantic Web [Berners-Lee et al. 94, Berners-Lee 99] aims to add a machine tractable layer to complement the existing web of natural language hypertext. This entails the addition of semantic metadata to web pages. However, given the very large number of already existing pages, the amount of effort that would be required to manually annotate them would be prohibitive. Because of this, the creation of semantic annotation, the linking of
web pages to ontologies, and the creation, evolution and interrelation of ontologies would only be feasible if they become automatic or semi-automatic processes.

This has led to efforts that use IE techniques to automate or support a manual process for producing semantic metadata [Kiryakov et al. 04], and for creating and enriching ontologies [Sabou et al. 05, Vargas-Vera et al. 07].

2.6 Portability Work – Motivation and Related Approaches

Tailoring an IE system to new requirements is a task that varies in scale dependent on the degree of variation in parameters such as style, genre, language, etc. Research in this area been one of the most fruitful areas of development since the end of the MUC programme in the late 1990s.

There are three broad currents of work in this area:

1. Learning extraction rules of models from annotated examples.
2. Embedding learning systems within end-user systems.
3. Supporting the development of rules/models by skilled specialist staff.

The former two fall under the heading of adaptive information extraction which is used to cover all methods that are based on using machine learning algorithms to automate the creation of IE systems. The latter refers to providing infrastructural support to developers in order to minimise the time and effort required for building a new system.

2.6.1 Adaptive Information Extraction

Most of the adaptive IE approaches involve supervised learning, where the results produced by humans for a particular task are collected in large quantities and used as inputs to learning algorithms. The collection of large quantities of training data is a major factor in the success of this type of work. The advantage is a reduction in the need for skilled staff to perform
system porting. The disadvantage, however, is that only simple data can be extracted, or complex data from simple texts. For instance, extracting scenario templates from seminar announcements is relatively easy as each text contains a single event of a pre-determined type – a seminar taking place; there is usually a single time expression representing the time of the seminar; the person names that appear are normally the speakers; and so on.

Another disadvantage of adaptive IE is that large volumes of training data are usually required, which are time consuming to create as they involve manual annotation of many example texts. In order to address this issue, researchers have created systems where the learning algorithm is embedded in the manual annotation tool and the training of the system is done in parallel with the annotation of the data. The advantage of this is that the system can learn while the manual annotation is performed, and it can start to assist the user in the annotation task as soon as it has enough data to reach a threshold level of confidence. This way, the system quickly learns the simpler cases, which it starts to annotate automatically, allowing the human annotator to concentrate on the more complex ones that require more training data. This type of systems are said to be using mixed-initiative learning [Day et al. 97] because the ‘initiative’ to annotate a piece of text is shared between the human annotator and the learning system. Good examples of mixed-initiative systems are the Alembic Workbench [Aberdeen et al. 95] and, more recently, the Melita system [Ciravegna et al. 03]. This approach addresses the problem of the costs of producing training data associated with the learning approach by speeding up the annotation process.

A further optimisation has been introduced by the means of active learning [Finn & Kushmerick 03], which lets the learning system choose which of the yet-unannotated documents is to be chosen next. This can speed up the training process by allowing the system to always select the document that is most likely to contribute to the improvement of the model that is being built.

A survey of approaches and systems that employ adaptive IE techniques is presented in [Turmo et al. 06]. It details the ways Machine Learning techniques are applied for Information Extraction, it lists the most popular
algorithms used for this type of tasks, and provides descriptions for a large number of systems in this area. Performance evaluation results are also presented for a selection of the tools described. The survey concludes by observing that it is still difficult to decide which technique is best suited for any given task or domain.

2.7 Summary

Information Extraction deals with extracting facts from unstructured text, usually producing filled-in templates or database records. Work in this area started in the 1970s, but the field has crystallised during the MUC series of conferences in the last years of the 20th Century, which were followed by the ACE conferences in the early 2000s.

The lack of portability of IE techniques was identified as an issue as early as 1995, with the MUC-6 conference calling for systems that are more re-usable and easier to re-target. In answer to this need, Machine Learning (ML) techniques started being applied to IE, giving rise to the sub-field of Adaptive Information Extraction. In Adaptive IE, the development effort changes from hand-crafting a set of heuristics to selecting the appropriate ML algorithm and finding a set of features that give good results for each of the sub-tasks. Improving the system performance depends on access to increasing amounts of training data.

While Machine Learning does help in addressing the issue of portability, it is not always the most appropriate solution. In some cases, training data can be difficult to obtain. For example, good quality training data may require manual annotation by highly specialised domain experts, who may be difficult to find, or whose time may be very restricted or costly. In such situations it may be more effective to elicit the domain knowledge through interviews with the domain experts, followed by encoding this knowledge into rules hand-crafted by language engineers. Unfortunately, other approaches to IE portability have not received the same amount of attention as Adaptive IE.
Chapter 3

Portable Information Extraction: Two Approaches

The first part of this chapter introduces JAPE – a formalism that supports the description and execution of rules for automatic annotation of text. The second part is dedicated to OLLIE – a system for human-machine collaborative annotation, backed by Machine Learning. JAPE and OLLIE, together with the infrastructural support described in Appendix A, give rise to an environment where portability of IE systems can be improved.

3.1 Knowledge Engineering and Machine Learning for Information Extraction

In terms of the methodology used, there are two basic approaches to the design of IE systems: Knowledge Engineering and Machine Learning.

The Knowledge Engineering approach is based on manually developed rules and grammars, created by knowledge engineers. If necessary, the engineers may consult domain experts who help them in understanding the problem that needs to solved, the types of information that should be extracted, and the required levels of performance. Typically, the developers have access to an annotated corpus of moderate size. A segment of this corpus is used as training data, and is seen as a model of the behaviour that the
final system must exhibit. The remainder of the annotated corpus is used as a gold standard against which the behaviour of the system is measured. The Knowledge Engineering approach benefits from the human developer’s expertise and intuition; an experienced engineer will be able to generalise from the examples seen in the training corpus and will attempt to find the optimal balance between general rules that cover many cases and very specific rules, that cover only a few. The disadvantage of the Knowledge Engineering approach comes from the amount of effort involved in manually crafting and fine-tuning the sets of rules. Historically, Knowledge Engineering was the first approach used for developing IE systems.

The Machine Learning approach aims to address the shortcomings of the Knowledge Engineering method by removing the effort needed to create and fine-tune the rule-set. In this case, an annotated corpus is also needed, which is again split into a training section and a gold standard one. The training corpus is used to automatically construct a model based on the examples seen. This model can be either symbolic (such as in the case of decision rules, or decision trees), vectorial (e.g. for algorithms based on support vector machines), or purely statistical (such as the one used by Bayesian methods). The training corpus used by the learning approach usually needs to be much larger than the one used by knowledge engineering approaches as in this case there is no human intuition or background knowledge. In order for the system to learn a particular pattern, it needs to be presented with explicit examples for each possible slight variation.

The learning approach changes the effort paradigm from one of manually developing and fine-tuning rules to one of creating large manually-annotated training corpora. Depending on the particular application setting, the knowledge engineering or the learning approach may be more appropriate.

3.2 Knowledge Engineering with JAPE

Knowledge engineering refers to human engineers prescribing the behaviour of the system using their knowledge and intuition, usually stipulated through some formalism that is both easy for the humans to author and for the machines to interpret and follow. These formalisms can take many shapes,
e.g. lists of rules to be executed in sequence, decision trees, unification grammars, regular expressions, etc., each having their own advantages and disadvantages.

JAPE is a formalism dedicated to the expression of rules for information extraction, supporting the building of Knowledge Engineering-based systems. The main requirements for such an expression mechanism are:

i It should be simple to write and understand. This allows rapid prototyping and encourages reuse, extensions and interchange of existing resources.

ii It should be expressive enough to model all the linguistic phenomena necessary for achieving the final aim, in this case, performing Information Extraction.

iii They should be efficient during the execution stage. Text processing algorithms can be very complex leading to long execution times, which affects the frequency with which experiments can be carried out. An efficient implementation is a definite advantage.

Having these considerations in mind, we chose to base our formalism on regular expressions. The following addresses each of the requirements in turn:

i Regular expressions are a well known mechanism, with which most engineers are already familiar. This reduces the gradient of the learning curve when familiarising with the new system. While the representation of the actual expressions used in the new formalism will not be identical with other systems (for reasons explained in detail later), the semantics associated with the expressions are the usual ones. This allows for an easy translation of eventual existing resources (e.g. previously created Perl programmes).

ii Previous experience in early IE systems, such as SRI’s TACITUS and FASTUS [Hobbs 91, Hobbs et al. 96] has shown that relatively shallow analysis, such as the one that can be performed through the use of regular expressions, is sufficient for Information Extraction.
iii Regular expressions and the closely-related finite state methods benefit from well-known optimisation methods for both space and time efficiency. They have been previously used successfully for various types of text processing and are known to be very efficient.

The typical finite state machines, either acceptors or transducers, previously used for text processing, use the characters in the text as the input alphabet. In our case, the input data comprises an annotation graph, and consequently we have to take a different approach. The rest of this section describes the JAPE formalism used for manual creation of text processing rules and the execution environment implemented for applying these rules.

3.2.1 JAPE – a Finite State Machine forAnnotations

JAPE is based on the regular expressions formalism, which it extends in order to address the fact that the data structure being used as input is not the usual sequence of items (e.g. a string of characters) but rather a directed acyclic graph (DAG) of annotations.

Typical regular expressions implementations, such as the ones used by the Perl programming language, process text directly using the characters as their input alphabet. This is not ideal in an IE context, because the input text is usually augmented with the results of preceding linguistic analysis such as morphological or syntactic information. In the context of a language engineering framework such as GATE, the simplest solution is a finite state machinery that is capable of running directly over the data model used internally by all processing resources, i.e. directly over annotations. JAPE is such a tool.

The central concepts in JAPE are those of grammar, phase and rule; a grammar is a sequence of phases and each phase is made up of a set of rules. These are described in simple plain text files which get then compiled into finite state transducers at initialisation time.

The structure of the rules is inspired from production rule systems: it has a left-hand-side (LHS), which describes patterns to be matched, and a right-hand-side (RHS), which defines actions to be taken upon successful match of the LHS patterns. An example of a JAPE rule is presented in Listing 3.1.
It matches a sequence of four numbers separated by dots, which might be

```
1 Rule: IPAddress
2 {
3  {Token.kind == number}
4  {Token.string == "."}
5  {Token.kind == number}
6  {Token.string == "."}
7  {Token.kind == number}
8  {Token.string == "."}
9  {Token.kind == number}
10 } : ipAddress
11 -->
12 : ipAddress.Address = {kind = "ipAddress"}
```

Listing 3.1: A simple JAPE rule

an Internet address, and generates a new annotation of type `Address` with
the only feature `kind` having the value `ipAddress`.

A manually authored IE system has at its core a large set of such rules which,
when run in the right sequence and benefiting from the output of previous
modules, recognise the required nuggets of information from the input texts.

### 3.2.2 Features of the JAPE Language

The syntax of the language used for writing JAPE grammars is inspired by
the Common Pattern Specification Language (CPSSL) [Appelt & Onyshkevych 96].
It provides means of defining patterns based on annotations and actions
for manipulating annotations on documents. It also has a mechanism for
specifying complex actions using the Java programming language with
direct access to the underlying GATE programming interface. This is useful
for implementing side-effects that manipulate the data structures at a lower
level.

#### Rules

A JAPE rule is composed of a preamble which is used to declare the rule
name and, if necessary, priority, a pattern side (left-hand-side – LHS) and
the action side (right-hand-side – RHS) separated by the sign `=>`.  

```
The patterns are specified in terms of annotation types and features. The basic element of a pattern looks like:

\{AnnotationType.featureName == featureValue, ...\}

All constraints included in a pattern element refer to the same annotation so the same annotation type needs to be used throughout. Here is an example that specifies a word that starts with an upper case letter (based on the output of the tokeniser):

\{Token.kind == "word", Token.orth == "upperInitial"\}

Several pattern elements can be combined in patterns by means of juxtaposition, disjunction and the ‘?’, ‘*’ and ‘+’ Kleene operators:

**juxtaposition:** Several pattern elements listed one after the other will match adjoining annotations that correspond to each pattern element in part.

**disjunction:** Several pattern elements or juxtapositions of pattern elements can be combined in a disjunction which will match a sequence of one or more annotations that match one of the branches of the disjunction.

‘?’: The *question mark* operator adds an optional pattern element or juxtaposition to a pattern.

‘+’: The *plus* operator causes the sub-pattern to which it is applied to match one or more times in sequence.

‘*’: The *star* operator causes the sub-pattern to which it is applied to match from zero to as many times as possible in sequence.

All operators apart from juxtaposition can only apply to pattern segments that are bracketed (included between ‘(‘ and ‘)’). Bracketed parts of the pattern can be bound to a label by adding a label with the syntax :<label name> after the closing bracket. All the annotations that are matched by a bound sub-pattern form the input to the actions on the RHS of the rule. A few examples of JAPE syntax elements are illustrated in Listing 3.2.
//This is a juxtaposition
{Token.string == "Mr"}{Token.orth == upperInitial}

//This is a disjunction
({Token.string == "Mr"} | {Token.orth == upperInitial})

//The '?', '+', and '*' operators
({Token.orth == upperInitial})?
({Token.orth == upperInitial})+
({Token.orth == upperInitial})*

//Labelled sub-patterns

({Token.string == "Mr"} |
{Token.string == "Mrs"}):title
({Token.orth == upperInitial})* :name
):fullName

Listing 3.2: Elements of JAPE patterns syntax

The RHS of each rule consists of a list of one or more actions that get executed when the pattern on the LHS has matched some segment of a document. The only action type defined by JAPE is one that generates a new annotation covering the document span indicated by the annotation or annotations that were matched in the LHS and bound to a label. For example, if the rule in Listing 3.1 were to match some input the result of the RHS action would be to generate a new annotation of type Address over-spanning all the annotations matched by the pattern in the LHS and nothing more. More complex actions can be implemented using the facility to include Java code as a RHS action.

Phases

A JAPE Phase is made up of a list of rules, a preamble containing various declarations and options and, optionally, a set of macro definitions.

A phase represents a set of rules that are intended to run at the same time. The arrangement of rules into phases is not random; all rules in the same phase compete for matching the input and this can lead to interaction between the rules.
Chapter III: Portable Information Extraction: Two Approaches

Matching Styles

Because several rules may match the same input segment and because different application contexts require different behaviours when this happens, JAPE allows the user to choose the kind of desired behaviour by means of the control option in the preamble of the phase. There are a few predefined behaviours or matching styles that JAPE implements and which are described next.

Appelt

The Appelt matching style is probably the most used one and is inspired from Doug Appelt’s TextPro system [Appelt & Martin 99], hence the name. It specifies that from all the rules that potentially match a part of the input starting from the same location in the annotation graph, the one with the longest match should be selected. If there are several rules that match the exact same part of the input (i.e. the length of the matching segment is the same) then the rule with higher priority is chosen. If there are still more than one rules that can match the same input and have the same priority, then the rule appearing first in the phase will take precedence. The intention behind the Appelt matching style is that for every piece of input, there can be at most one rule that is fired. It is possible that the same rule can match the same document extent by traversing different annotation sequences. This can happen when the rule makes use of disjunction or when multiple annotations of the same type cover the same document segment. In this cases, JAPE will choose one of the possible matches to be applied and ignore the others. If the debug mode is enabled, a warning will be displayed whenever this occurs. It should be noted however, that in such cases the final result is usually the same – the same rule applies to the same document extent, regardless of which particular annotations are matched. The final outcome is only changed if side-effects are employed in the rule actions. If this situation is undesirable, it can usually be avoided by either redesigning the affected rule or by changing the behaviour of the upstream components to not doubly-annotate the same text segment.

Once a rule is fired (i.e. the actions on the RHS are run) the input matched by its LHS is considered “consumed” and the execution of the phase resumes from where the match ended in the annotation graph.
This rule application style is the most usual in information extraction applications where the longest match is usually preferred; when several named entities are contained within each other, it is usually the longest that is salient. For example, Sheffield should not be recognised as a location within the string University of Sheffield, which is itself the name of an organisation.

**Brill**

The Brill matching style (named after Eric Brill as it is similar to the way his transformation-based rule learning systems work [Brill 94, Brill 95]) differs from Appelt in the fact that, starting from a given position in the annotation graph, all rules that match some segment of the input are fired. All the input that was matched by at least one fired rule is considered consumed and the matching for the phase continues from where the longest match finished.

**All**

The All matching style is similar to Brill, in the sense that it also fires all rules that match from the current starting position in the document. The difference is in the way the input is consumed: while Brill consumes all matched input and continues the matching process from the point where the longest match ended, the All style simply moves the current position to the next offset in the document. See Figure 3.1 for a graphical representation that illustrates the difference between the various matching styles, including Brill and All.

The Brill and All rule application styles are useful when all possibly valid interpretations of a document segment need to be identified. Typically this will create spurious annotations that are normally filtered by subsequent phases that apply contextual rules.

**First**

The First matching style is, in a way, the opposite of Appelt in the sense that the first rule that successfully satisfies its LHS pattern by matching a part of the input will be fired and the corresponding part of the input will be consumed. Rule priorities or position of rules in the phase are not considered, so if there are several rules that may match at any given location in the annotation graph, there is no way to control which one will be fired. The matching algorithm normally tries to match each rule in turn and advance
over the input until all matching stops. However, when the First style is used, it will stop the first time it encounters a successful match and there is no way to control the order in which the rules will be selected for matching attempts (this depends on several hashing functions which are based on the actual values of memory pointers, leading to random orderings).

This application style can be useful when the application developer is not interested in finding alternative interpretations – the first time that a text segment can successfully be matched to a rule, this is applied and no further matching attempts are made for the same document extent. This is obviously more efficient that using the Appelt style and can be appropriate for particular entity types.

**Once**

The Once matching style is similar the the First one in the fact that it will stop the first time a successful match is possible. Additional to that, it will also cease the execution of the current phase upon the successful application of any of the rules contained in the phase.

This is useful for rules that are used mainly for their side-effects. One example is a rule that tries to determine the genre of a document – once it has enough information to cause a successful match, it will set the value of a particular feature of the document and the phase will exit. Further applications of the same rule set over the same document would not be required.

The effect of choosing one of the different matching styles is illustrated in Figure 3.1. The top row represents the annotations on the input document; in this case we assume a sequence of four annotations of type A. The lower part of the figure shows which matches would occur when using each of the possible matching styles, for a JAPE phase containing a single rule with a pattern of “(\{A\})\*”. Each of the thick horizontal bars represents a match.

**Ingestion of Input**

As mentioned, the input for JAPE rules comprises the annotations over documents. Sometimes, for efficiency reasons or for the sake of rule simplicity, it is useful to be able to select only some of the annotations for processing
with JAPE. This is achieved using the Input declaration in the preamble of the phase.

An Input declaration consists of a list of annotation types. Its effect is that all the annotations of types not listed are ignored by the JAPE matching algorithm – it is as if they did not exist. To fully explain how this will affect the matching process it useful to look in more detail at how JAPE traverses the annotation graphs.

The annotations in an annotation set (one of the possible several layers of annotation data associated with a document) form a directed acyclic graph. Each annotation has a start offset and an end offset in the character sequence that makes up the document content. These define nodes of the graph while the annotations themselves can be seen as arcs. The JAPE matching algorithm will essentially try to find a path from one node to another by only passing through annotations that match the patterns of the rules. Running
a JAPE phase over an annotation graph means travelling from the node with the lowest offset to the one with the highest while firing actions whenever rules match on the way. The various matching styles will influence which path gets chosen or how many of the parallel paths between the same two nodes are used.

It is noteworthy that the annotation graph is not guaranteed to be connected – there may not be a path from the node with the lowest offset to the one with the highest. This requires JAPE to deal with gaps in the input, otherwise some documents will only be partially processed (only the connected component containing the starting node will be processed out of the entire annotation graph). To address this issue, JAPE skips over the gaps, essentially joining the node with the highest offset from one connected component with the one having the lowest offset in the next connected component of the graph.

Figure 3.2: A path through overlapping annotations.

Returning to the Input declaration, Figure 3.2 can be used to illustrate the effect of input filtering on the JAPE rules. If no filtering is performed, then a pattern like `{Person}{Token}` (i.e. a Person annotation followed by a Token annotation) will fail, as the `SpaceToken` annotation between the second and third Token annotations will intervene between the Person and the Token annotations. However, if the `SpaceToken` annotations are not included as input, then the pattern would succeed because of the way JAPE joins one annotation to the next. The `Person` annotation will be followed by the `Token` one as far as JAPE is concerned. Similarly, if only `Person` and `Location` annotations were to be included as input, then a pattern of `({Person})+{Location}` would successfully match the two `Person` annotations followed by the `Location` one, consuming all the input.
Phase: NP
Input: Token
Options: control = appelt

Macro: NOUN
( {Token.category == NN} | {Token.category == NNS} |
{Token.category == NNP} | {Token.category == NNPS} |
{Token.category == NP} | {Token.category == NPS} |
{Token.category == CD} | {Token.category == PRP} |
{Token.category == WP} )

Macro: DET
( {Token.category == DT} | {Token.category == PRP} |
{Token.category == WDT} )

Macro: ADJ
( {Token.category == JJ} | {Token.category == JJR} |
{Token.category == JJS} )

Macro: RB
( {Token.category == RB} | {Token.category == RBR} |
{Token.category == RBS} )

Macro: CC
( {{Token.category == CC} | |
  ( {{Token.string==''} | |
    {Token.string==''} | |
    {Token.string==n}} |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
  ) |
Grammars

A JAPE grammar is a cascade of one or more phases that will be run in sequence. The same phase can be listed repeatedly in the cascade if it is required to run several times. Grammars are usually used as logical units, grouping together a set of phases that perform a given function.

Beside listing the phases, grammars are also the visibility domain for JAPE macros. A macro is a piece of JAPE code associated with a name. Following its declaration, a macro name can be used to replace sections of JAPE rules that occur repeatedly. Although macros are defined by phases, they are actually associated with the grammars – once defined, a macro is visible in all the succeeding phases.

Listing 3.3 shows the full listing of a simple JAPE phase that finds noun phrases based on the output of the part of speech tagger. It shows most of the JAPE elements previously discussed, described here by line numbers:

line 1: the name of the phase.

line 2: the Input declaration. This JAPE phase only uses Token annotations.

line 3: options for the current phase. In this case the matching style is specified as Appelt.

lines 5 – 31: define macros that can be used in the current phase and all the subsequent ones. They are used here for the benefit of readability when writing the rule patterns.

line 33: the start of a JAPE rule, including the name of the rule.

line 34: declaration for the rule priority.

lines 35 – 38: the pattern side (LHS) of the rule. Note the use of Kleene * operators, a disjunction, and a label.

line 39: the separator between the pattern and action sides of the rule.

line 40: the action of the rule. It creates a new annotation of type NP covering the same document extent as covered by all the annotations.
matched from the input. The new annotation will have a feature rule
with value NP1.

3.2.3 Support for Ontologies in JAPE

As described in Section A.7, GATE provides support for using ontologies as
a mechanism for data modelling and storage of results. JAPE can benefit
from gaining access to the more complex hierarchical matching algorithms
provided by ontologies which can, in some cases, be used to replace the sim-
ple exact matching normally used. Combining the power of ontologies with
JAPE’s pattern matching mechanisms can simplify the creation of applica-
tions. This section describes how taxonomic data can be used to augment
the power of JAPE, and it includes an example of such an application.

In order to use ontologies with JAPE, one needs to load an ontology in
GATE before loading the JAPE transducer. Once the ontology is known to
the system, it can be set as the value for the optional ontology para-
meter for the JAPE grammar. Doing so alters slightly the way the matching
occurs when the grammar is executed. If a transducer is ontology-aware
(i.e. it has a value set for the ‘ontology’ parameter) it will treat all oc-
currences of the feature named class differently from the other features of
annotations. The values for the feature class on any type of annotation
will be considered to be the names of classes belonging the ontology, and
the matching between two values will not be based on simple equality but
rather hierarchical compatibility. For example, if the ontology contains a
class named ‘Politician’, which is a sub class of the class ‘Person’, then a
pattern of {Entity.class == "Person"} will successfully match an anno-
tation of type Entity with a feature class having the value “Politician”.
If the JAPE transducer were not ontology-aware, such a test would fail.

This behaviour allows a larger degree of generalisation when designing a set
of rules. Rules that apply to several types of entities mentioned in the text
can be written using the most generic class they apply to, and need not
be repeated for each subtype of entity. One could have rules applying to
Locations without needing to know whether a particular location happens
to be a country or a city. If a domain ontology is available at the time of
building an application, using it in conjunction with the JAPE transducers can significantly simplify the set of grammars that need to be written.

The ontology does not normally affect actions on the right hand side of JAPE rules, but when Java is used on the right hand side, then the ontology becomes accessible via a local variable named \texttt{ontology}, which may be referenced from within the right-hand-side code.

In Java code, the \texttt{class} feature should be referenced using the static final variable, \texttt{LOOKUP\_CLASS\_FEATURE\_NAME}, that is defined in the Java interface \texttt{gate\_creole\_ANNIEConstants}.

**Populating Ontologies**

One typical application that combines the use of ontologies with NLP techniques is finding mentions of entities in text. The scenario is that one possesses an existing ontology and needs to use Information Extraction to populate it with instances whenever entities belonging to classes in the ontology are mentioned in the input texts. This process is typically named \textit{Ontology Population}.

Other applications linking ontologies and Information Extraction methods include:

**Semantic Annotation:** is the process of identifying mentions of entities in input text. This is essentially an evolution of the typical IE task, where the set of annotation types to be created is replaced with a taxonomy of classes and their instances. Each instance of a class represents an entity, e.g. \textit{London} could be an instance of the class \textit{City}, which is itself a sub-class of \textit{Location}. The task is one of annotating all occurrences of entity names in the input text.

This application of IE techniques emanated from the Semantic Web area and became a popular method of generating semantic mark-up that can be used as input by the tools developed in that research field. See e.g. [Kiryakov et al. 04] for a more comprehensive discussion regarding the use of Semantic Annotation to create metadata for the Semantic Web.
In many settings, the semantic annotation process produces, as a side effect, data that can be used for ontology population. One such example is the KIM platform [Popov et al. 04] which makes use of semantic annotation to create metadata for documents and to augment its internal knowledge base.

Ontology Learning: comprises the creation of new ontologies, or the extension of existing ones with new taxonomic information (new classes or properties) as a result of analysing input text. As this is a very difficult task, the methods used are usually semi-supervised. See e.g. [Brewster et al. 02] for a description of a system, named Adaptiva, that performs this task using a tight loop of interactions between the user and the system. The work presented in [Ahmad & Gillam 05], and [Sabou 04] shows that, when applied to a restricted language (e.g. the language of a specialised domain, respectively software documentation), Ontology Learning can be performed largely automatically resulting in some intermediate representation that can then be manually validated to create an ontology.

Let us turn our attention to the task of ontology population. We assume that, as a result of a semantic annotation process, we have text that has been marked up with annotations of type ‘Mention’ having a feature ‘class’ specifying the class of the entity mentioned. The task we are seeking to solve is to add instances in the ontology for every Mention annotation.

The example presented in Listing 3.4 is based on a JAPE rule that uses Java code on the action side in order to access directly the ontology API.

This will match each annotation of type Mention in the input and assign it to a label ‘mention’. That label is then used in the right hand side to find the annotation that was matched by the pattern (lines 5–10); the value for the class feature of the annotation is used to identify the ontological class name (lines 12–14); and the annotation span is used to extract the text covered in the document (lines 16–21). Once all these pieces of information are available, the addition to the ontology can be done. First, the right class in the ontology is identified using the class name (lines 24–29) and then a new instance for that class is created (lines 30–31).
Chapter III: Portable Information Extraction: Two Approaches

Listing 3.4: Populating ontologies with JAPE

```java
Rule: FindEntities
({Mention}): mention
-->
{
    // find the annotation matched by LHS
    // we know the annotation set returned
    // will always contain a single annotation
    Annotation mentionAnn = (Annotation)
        ((AnnotationSet)bindings.get("mention").
            iterator().next());

    // find the class of the mention
    String className = (String) mentionAnn.getFeatures().get( gate.creole.ANNIEConstants.LOOKUP_CLASS_FEATURE_NAME);

    // find the text covered by the annotation
    String mentionName = doc.getContent().
        getContent( mentionAnn.getStartNode().getOffset(),
            mentionAnn.getEndNode().getOffset() ).
        toString();

    // add the instance to the ontology
    // first identify the class
    TClass aClass = ontology.getClassByName( className );
    if( aClass == null ){
        System.err.println( "Error class " + className
            + " does not exist!" );
    }

    // now create the instance in the ontology
    ontology.addInstance( mentionName, (OClass)aClass );
}
```

The solution presented here is purely pedagogical, as it does not address many issues that would be encountered in a real life application solving the same problem. For instance, when an entity mention is identified in the text, the application would have to check whether the entity mentioned is already known to the ontology, and only add a new instance when it is not found. Also it is naïve to assume that the name for the entity would be exactly the text found in the document. In many cases, entities have several aliases – for example, the same person name can be written in a variety of forms depending on whether titles, first names, or initials are used. A process of name normalisation would probably need to be employed in order to make
sure that the same entity, regardless of the textual form it is mentioned in, will always be linked to the same ontology instance.

### 3.2.4 Initialisation of a JAPE Grammar

Upon loading a JAPE grammar, each phase is converted into a *finite state machine* (FSM) that accepts the annotation chains described by the rules’ pattern sides.

The concepts of *regular expression* and *finite state machine* have been used widely in computer science literature and they have several, largely equivalent, definitions. In the interest of clarity we shall start by providing definitions for them as they are used in this thesis.

**Definition 3.1.** A *regular expression* is a syntactic formalism for representing languages in a compressed manner. Given a set of input symbols $\Sigma$, a language is a set of symbol sequences, or strings. If $r$ is a regular expression, then let $L(r)$ be the language denoted by $r$. Regular expressions over the alphabet $\Sigma$ are defined by the following set of rules:

1. $\epsilon$ is the regular expression that denotes $\{\epsilon\}$, i.e. the set that contains only the empty string.
2. if $a \in \Sigma$, then $a$ is the regular expression that denotes $\{a\}$, i.e. the set that contains only the string $a$.
3. if $r$ and $s$ are regular expressions, then:
   
   (a) $(r)$ is the regular expression that denotes $L(r)$. In other words, parentheses can be added around regular expressions without changing their meaning.
   
   (b) $rs$ is the regular expression that denotes $L(r) \times L(s)$, i.e. the language in which each string is formed by the concatenation of a string from $L(r)$ and one from $L(s)$.
   
   (c) $r|s$ denotes $L(r) \cup L(s)$.
   
   (d) $r^*$ denotes $L(r)^*$, i.e. $\{\epsilon\} \cup L(r) \cup (L(r) \times L(r)) \cup (L(r) \times L(r) \times L(r)) \cup \ldots$
Chapter III: Portable Information Extraction: Two Approaches

For convenience, we also introduce the following shorthand notations:

1. \( r? \) is the same as \( r|\epsilon \).
2. \( r^+ \) is the same as \( rr^* \).

Definition 3.2. A non-deterministic finite automaton (NFA) is defined as a mathematical model consisting of:

1. a set of states \( S \),
2. a set of input symbols \( \Sigma \) (the input alphabet),
3. a transition function \( trans : S \times \Sigma \rightarrow S^2 \) that maps \(<\text{state, symbol}>\) pairs to sets of states,
4. a state \( s_0 \) that is distinguished as the start (or initial) state,
5. a set of states \( F \) that are distinguished as final states.

An NFA is said to accept a string of input symbols \( \sigma_0 \sigma_1 .. \sigma_n \) if

\[
(\exists s_1 s_2 .. s_n s_{n+1}, \forall i \in 0..n, s_{i+1} \in trans(s_i, \sigma_i)) \land s_{n+1} \in F
\]

NFAs can be represented graphically as directed graphs where the nodes correspond to NFA states while the arcs are annotated with input symbols and represent the transition function. The symbols used are presented in Figure 3.3.

Theorem 3.1. For any arbitrary regular expression, it is possible to construct a NFA for which the set of accepted strings is the same as the language denoted by the regular expression.

The proof for this theorem is well known. We are including it here due to its constructive character – it essentially describes how to automatically build the equivalent NFA for any regular expression.
Proof. In the following graphical depictions we shall use $\epsilon$ as a representation for the NFA corresponding to the $r$ regular expression. The state to the left is the initial state of the NFA, while the state to the right is the final state. If the original NFA contains multiple final states, we modify it by marking all final states as non-final, constructing a new final state, and adding $\epsilon$-labelled transitions from each of the original final states to the newly created one. The new state becomes the only final state. This modification does not change the language accepted by the NFA. Let us consider in turn all types of regular expressions as described in the definition.

1. We associate with the regular expression $\epsilon$ the following NFA:

   \[ \begin{array}{c}
   \text{State} \\
   \text{Start state} \\
   \text{Final state}
   \end{array} \begin{array}{c}
   - \epsilon \\
   - \\
   \odot
   \end{array} \]

   The language described by the regular expression is the same as the accepting set of the NFA, i.e. $\{\epsilon\}$.

2. We associate with the regular expression $a$ the following NFA:

   \[ \begin{array}{c}
   \text{State} \\
   \text{Start state} \\
   \text{Final state}
   \end{array} \begin{array}{c}
   - a \\
   - \\
   \odot
   \end{array} \]

   Starting with the initial state of the NFA, the only possible transition is to the only final state, while consuming the string $a$. Thus the accepting set of the NFA is $\{a\}$, which is the same as the language described by the regular expression.

3. Given the regular expressions $r$ and $s$ and their associated NFAs $\begin{array}{c}
   \text{State} \\
   \text{Start state} \\
   \text{Final state}
   \end{array} \begin{array}{c}
   - r \\
   - \\
   \odot
   \end{array}$, respectively $\begin{array}{c}
   \text{State} \\
   \text{Start state} \\
   \text{Final state}
   \end{array} \begin{array}{c}
   - s \\
   - \\
   \odot
   \end{array}$:

   (a) The regular expression $(r)$ is associated with the following NFA:

   \[ \begin{array}{c}
   \text{State} \\
   \text{Start state} \\
   \text{Final state}
   \end{array} \begin{array}{c}
   - r \\
   - \\
   \odot
   \end{array} \]
The accepting set of the LFA is $L(r)$ which is by definition the same as $L((r))$.

(b) The regular expression $rs$ is associated with the following NFA:

Let $t \in L(rs)$. Then $t = t_r \cdot t_s$, $t_r \in L(r)$, $t_s \in L(s)$. Starting from the initial state, the NFA can only transition to state 1, without consuming any input, then it can consume $t_r$ while transitioning to state 2, from where it can only move to state 3, without consuming any input. Starting from state 3, the NFA can consume $t_s$, while moving to state 4, from where it can only move to final state, with no input. Consequently a sequence of transitions from the start state to the final one is possible while consuming the string $t$, hence $t$ is included in the accepting set of the NFA.

Conversely, let $t$ be a string in the accepting set of the NFA. Due to the topology of the NFA, $t = \epsilon \cdot t_r \cdot \epsilon \cdot t_s \cdot \epsilon = t_r \cdot t_s$, where $t_r \in L(r)$, $t_s \in L(s)$. From the definition of regular expressions it follows that $t \in L(rs)$.

(c) The regular expression $r|s$ is associated with the following NFA:

Let $t \in L(r|s)$, then, by definition, $t \in L(r) \cup L(s) \Rightarrow t \in L(r) \lor t \in L(s)$. If $t \in L(r)$ then $t$ is also in the accepting set of the NFA, as the NFA can move with no input from the initial state to state 1, then consume $t$ while transitioning to state 2 and from there to the final state. Similarly if $t \in L(s)$, $t$ is also in the accepting set of the NFA.

Conversely, let $t$ be a string from the accepting set of the NFA. This implies there is a sequence of transitions from the start state to the final one that consumes $t$. The only states that can follow the start state are 1 or 3. If state 1 is reached, then the NFA can
only reach the final state by transitioning to state 2 which implies that \( t \in L(r) \). Similarly, if state 3 is reached, then \( t \in L(s) \). In other words, \( t \in L(r) \lor t \in L(s) \), which implies, by definition, that \( t \in L(r|s) \).

(d) The regular expression \( r^* \) is associated with the following NFA:

\[
\begin{array}{c}
\text{\eps} \\
\begin{array}{c}
\text{\eps} \\
\text{r} \\
\text{\eps}
\end{array}
\end{array}
\]

Let \( t \in L(r^*) \Rightarrow t \in L(\eps) \cup L(r) \cup L(r) \times L(r) \cup \ldots \). If \( t = \eps \) then \( t \) is accepted by the associated NFA due to the direct \( \eps \)-labelled transition from the initial to the final state. Otherwise \( t \in \prod_{n \geq 1} L(r) \), which means that \( t \) is a concatenation of \( n \) strings from \( L(r) \). It can easily be proven by induction that \( \forall n \geq 1, t \) is accepted by the NFA.

Conversely, let \( t \) be a string accepted by the NFA. This means that there is a sequence of states that consumes \( t \) and ends in the final state. If we observe this sequence of states in inverse order, we note that each occurrence of the final state must be preceded by an occurrence of the initial state (the final state is only reachable via paths that include the start state). Each sequence of states between consecutive occurrences of the initial and final states consumes either \( \eps \) or a string from \( L(r) \). Similarly, we can observe that each occurrence of the start state is either the first in the sequence or is directly preceded by the final state. Consequently the whole sequence of states that accepts \( t \) is a concatenation of one or more sub-sequences, each of which accepts either \( \eps \) or a string from \( L(r) \). Results that \( t \in \prod_{n \geq 1} (\{\eps\} \cup L(r)) = L(r^*) \).

The rules described above cover all cases included in the definition for regular expressions.

When a JAPE grammar is loaded in GATE, each phase is converted into a
finite state machine, a process that has several stages. Each rule is treated as a regular expression using annotation-based patterns as input symbols. A JAPE phase is a disjunction of rules, so it is also a regular expression. The first stage of building the associated FSM for a JAPE phase is the construction of a non-deterministic finite-state automaton, following the algorithm described in the proof of Theorem 3.1, which is largely equivalent to the one included in [Aho et al. 86].

Additional to standard regular expressions, JAPE rules also contain bindings (labels associated to pattern segments). These are intended to be associated to the matched input symbols (i.e. annotations) during the matching process, and are used while executing the actions caused by the rule firing. Upon creating the equivalent FSM for a given JAPE rule, bindings are associated with the FSM transitions. This changes the semantics of a transition – besides moving the state machine into a new current state, a transition may also bind the consumed annotation(s) with one or more labels.

In order to optimise the execution time during matching (at the expense of storage space), NFAs are usually converted to Deterministic Finite State Automata (DFAs) using e.g. the subset algorithm [Aho et al. 86]. In the case of JAPE this transformation is not possible due to the binding labels: two or more transitions from the NFA that match the same annotation pattern cannot be compacted into a single transition in the DFA if they have different bindings. Because of this, JAPE grammars are represented as non-deterministic finite state machines. A partial optimisation that eliminates the ε-transitions from the NFA is however performed.

The actions represented on the right hand side of JAPE rules are converted to compiled Java classes and are associated with final states in the FSM. The final in-memory representation of a JAPE grammar thus consists of a non-deterministic finite state machine, with transitions that use annotation-based patterns as input symbols, additionally marked with bindings information and for which the final states are associated with actions.

Starting from the following two JAPE rules:

Rule: PersonPrefix

(}
the associated NFA is constructed, as illustrated in Figure 3.4. Note that due to the fact that the final states are associated with different actions, they cannot be joined into a single one and are kept separate. This automaton is then optimised by eliminating the $\epsilon$-transitions, resulting in the NFA presented in Figure 3.5. For the sake of simplicity, the annotation patterns used are the most basic ones, depending solely on annotation type. In the graphical representation, the transitions are marked with the type of annotation that they match and the associated binding in square brackets.

![Diagram of a non-deterministic finite state machine compiled from JAPE rules.](image)

Figure 3.4: Example of a non-deterministic finite state machine compiled from JAPE rules.

It can be observed in Figure 3.5 that there are two transitions starting from state 1 (leading to states 2, respectively 4) that both consume annotations of type `Token`, thus even the optimised finite state machine is still non-
3.2.5 Execution of JAPE Grammars

The execution of a JAPE grammar can be described in simple terms as finding a path through an annotation graph where all the annotations traversed form a sequence that is accepted by the finite state machine built during the initialisation phase. The actual process is somewhat more complex than that, as it also needs to take into account the various matching modes, the filtering of input annotation types, to deal with the assignment of matched annotation to bindings, and to manage the execution of actions whenever successful matches occur.

Executing a JAPE grammar involves simulating the execution of a non-deterministic finite state automaton (NFA) while using an annotation graph as input. At each step we start from a document position (initially zero) and a finite state machine in a given state (initially the start state). Annotations found at the given document position are compared with the restrictions encoded in the NFA transitions; if they match, the annotations are consumed and the state machine moves to a new state. Ambiguities are possible at each step both in terms of input (several matching annotations can start at the
same offset) and in terms of available NFA transitions (the state machine is non-deterministic, so multiple transitions with the same restrictions can be present). When such ambiguities are encountered, the current state machine is cloned to create as many copies as necessary, and each such copy continues the matching process independently. The JAPE executor thus needs to keep track of a family of state machines that are running in parallel – henceforth we shall call these FSM instances.

Whenever one of the active FSM instances is moved to a new state, a test is performed to check if the new state is a final one. If that is the case, the FSM instance is said to be in an accepting state, and a copy of its state is saved for later usage.

When none of the active FSM instances can advance any further, the stored accepting FSM instances are used to execute JAPE actions, according to the declared matching style of the current grammar (see Section 3.2.2 for details).

A high-level view\(^1\) of the algorithm used during the execution of a JAPE grammar is presented in Listing 3.5, in a Java-inspired pseudo-code.

The next paragraphs contain some more detailed comments, indexed using the line numbers in the listing:

**line 1** The annotations present in the document are filtered according to the `Input` declaration in the JAPE code, if one was present. This causes the JAPE executor to completely ignore annotations that are not listed as valid input.

**lines 2–4** The matching process is initialised by setting the document position to 0, and creating empty lists of active and accepting FSM instances.

**lines 5–29** The matching continues until all the document text is exhausted.

**line 8** Each step starts from the current document position with a single FSM instance.

\(^1\)The view of the algorithm presented here is greatly simplified, for the sake of clarity. The actual implementation consists of a few thousand lines of Java code.
Chapter III: Portable Information Extraction: Two Approaches

Listing 3.5: JAPE matching algorithm

```java
processInputfilters();
currentDocPosition = 0;
activeFSMInstances = new List<FSMInstance>();
acceptingFSMInstances = new List<FSMInstance>();
while (currentDocPosition < document.length()){
    // create an initial FSM instance, starting from
    // the current document position
    activeFSMInstances.add(
        new FSMInstance(currentDocPosition));
    // advance all FSM instances,
    // until no further advance is possible
    while (!activeFSMInstances.isEmpty()){
        FSMInstance aFSM = activeFSMInstances.remove(0);
        // annotations to binding labels, as required;
        // create cloned copies as necessary and add them to
        // activeFSMInstances;
        // save any accepting state of aFSM
        // into acceptingFSMInstances;
    }
    if (!acceptingFSMInstances.isEmpty()){
        // execute the action(s)
    }
    // move to the new document position, in accordance
    // with the matching style.
}
```

**lines 12–22** While there are still active FSM instances, they are advanced as far as possible. Whenever ambiguities are encountered, cloned copies are created and added to the list of active FSM instances. Whenever an FSM instance reaches a final state during its advancing, a copy of its state is saved to the list of accepting FSM instances.

**lines 23–25** This segment of code is reached when there are no more active FSM instances – all active instances were advanced as far as possible and either saved to the accepting list (if they reached a final state during that process) or simply discarded (if they could advance no further but they still have not reached a final state).

At this point, any successful matches that occurred need to be acted
upon, so the list of accepting FSM instances is inspected. If there are any, their associated actions are now executed, according to the desired matching style. For instance if the matching style used is Appelt, then only the accepting FSM instance that has covered the most input will be executed; conversely, if the matching style is Brill, then all accepting FSM instances will have their actions executed, etc.

line 27 When this point is reached, all possible matches from the current document position were found and the required action executed. The next step is to move to the next starting position in the document, and re-start the matching process from there. Depending on the matching style selected, the new document position is either the oldPosition + 1, in the case of All, or matchingEndPosition + 1 in all other cases.

Time Complexity for JAPE

JAPE is similar to a programming language in the sense that it executes a set of user-created rules. As such, it is difficult to estimate its efficiency directly, as the execution time will depend on the intricacy of the rule set and the difficulty of the task they are addressing. To provide a good indication of the time complexity, we shall estimate an upper and lower bound, and then we shall proceed to perform an empirical evaluation to measure the actual performance in a particular setting.

There are many parameters that affect the time taken to execute a JAPE grammar, e.g. the number of rules in the grammar, the length of the input document, the average number of overlapping annotations, the average number of features for each annotation. For the remainder of this discussion, we shall:

- Assume that attempting to match one input annotation (whether successfully or not) is a constant time operation (i.e. \( \in O(1) \)).
- Define the length of the input document as the length of the longest possible sequence of annotations sorted by offsets (where the end offset of one annotation is less or equal to the start offset of the next). We shall denote this value as “n”.

63
• Define the level of ambiguity of the input document as the cardinality of the largest set of overlapping annotations (i.e. a set of annotations such that \( \exists o \), an offset that is greater than or equal to the start offset and less than or equal to the end offset of every annotation in the set.) We shall denote this value as “m”. Together, \( n \) and \( m \) describe the input document.

• Use the notation \( T(n, m) \) for the time complexity function, where \( n \) and \( m \) are as defined above.

\[
\text{Time Complexity Upper and Lower Bounds}
\]

In order to determine an upper bound for \( T(n, m) \) we need to estimate the maximum number of attempted matches that can occur. For this we shall construct an artificial example where all annotations are of the same type (A) and consider a JAPE pattern of “(\{A\})\^\ast”, that is executed using the All matching style. In this case, all attempted matches will be successful so we can evaluate the total number of attempted matches as the sum of the lengths of all successful matches, where the length is expressed as the number of A annotations covered. With these assumptions, the annotations on the input document would form a structure like that depicted in Figure 3.6, which is described by the values of the \( n \) and \( m \) parameters.

Let us start with a particular case, where \( m = 1 \). Starting from each document position, the match lengths will include 1, 2, 3, \ldots, up to the end of the document. However, each match longer than 1 is actually created by adding
Figure 3.7: Match count for \( m = 1 \)

a newly matched annotation to an existing previous match. Figure 3.7 shows all the matches that occur, highlighting the newly-matched annotations in each case. For each starting position, the total number of newly matched annotations is shown. It follows that:

\[
T(n, 1) = \sum_{i=1}^{n} i = \frac{n^2 + n}{2} \Rightarrow T(n, 1) \in O(n^2)
\]

This indicates that for non ambiguous documents, where there are no overlapping annotations that can generate alternative matches, the JAPE matching algorithm is of polynomial (quadratic) time complexity.

A typical example of this worst case is a rule that attempts to match a pattern of \( \{A\} \ast \{B\} \), using the \texttt{Appelt} matching mode, on an input that consist of a sequence of \{A\} annotations. This rule will match nothing (as there is no ending \( B \) annotation), but will attempt all possible matches starting from each possible document position before giving up. Because of this, the use of the Kleene star operator is discouraged. In many cases the same result can be obtained using a sequence of ? operators, where the maximum length of the matched sequence is limited (e.g. replacing \( \{A\} \ast \)
Chapter III: Portable Information Extraction: Two Approaches

with \((\{A\})?\{\{A\}\}?\ldots(\{A\})?)\).

The fact that JAPE runs in polynomial time is relevant because it shows that, even in pathological cases, where the rule set is highly ambiguous, using patterns that are not very discriminating, JAPE remains tractable as long as the input document is not too ambiguous.

Let us now return to the general case, where \(m\) has an arbitrary value. Figure 3.7 will change in the sense that, at every level, instead of \(i\) matches, we will now have \(m^1 + m^2 + \ldots m^i\) matches (as there are \(m\) alternative annotations for each position). It follows that:

\[
T(n,m) = \sum_{i=1}^{n} \sum_{j=1}^{i} m^j = m^n + 2m^{n-1} + \ldots + nm^1
\]

\[
= \frac{m(m^{n+1} - m)}{m-1} - mn \quad (\forall m > 1)
\]

\[
T(n,m) \in O(m^n)
\]

This shows that the JAPE execution time is greatly affected by the level of ambiguity of the input document. For documents that are completely ambiguous, i.e. documents that have alternative interpretations for every possible annotation, JAPE has an exponential time complexity, and so becomes intractable for any large examples. This is not surprising given that even the task of traversing all possible sequences of input annotations (without attempting any matches) requires exponential time in such a case.

The lower bound for the time complexity is \(O(1)\), as a grammar using the \texttt{Once} matching mode will terminate as soon as it has matched one annotation.

**Empirical Evaluation of Time Complexity**

We have shown above that \(O(1) < T(n,m) < O(m^n)\). While this is correct, it is not very informative due to the wide distance between the two bounds. To get a better indication of JAPE’s time complexity we have performed an experiment where we measured the actual execution time of a reasonably complex JAPE rule-set.
For this, we chose the set of grammars that are part of the ANNIE IE system, which include 20 individual phases totalling 169 rules. Of the 20 phases, 19 use the Appelt mode, while the remaining one uses Brill.

![Figure 3.8: JAPE execution time in ANNIE](image)

The experiment we performed consisted of executing the grammars over documents of increasing length, from 10 to 500 thousands of characters, in increments of 10,000. For each such document we calculated the total execution time by summing up the execution time for all of the 20 individual grammars. The results are represented as a graph in Figure 3.8.

It is easy to note that the execution time appears to increase linearly with the size of the text. This shows that even though the time complexity can reach exponential orders of magnitude, in many cases, including not trivial ones, JAPE offers an efficient mechanism for performing text analysis tasks, in particular information extraction.

### 3.3 Machine Learning with OLLIE

The previous section was dedicated to improving portability of IE systems built around the knowledge engineering approach. The other main approach, Machine Learning, is the focus of this section, where we present OLLIE – a system supporting On-Line Learning for Information Extraction [Tablan et al. 03].
OLLIE is a web-based system for human-computer collaborative annotation of textual data. It supports the manual annotation of textual documents inside a web browser. The manually-created annotations are used as training data for machine learning algorithms that run on the server. The models created from the training data are then used for pre-annotating new data, which helps speed-up the manual annotation process. As the cycle continues, the trained models become more precise, they make fewer mistakes, requiring less manual intervention. The output of this process comprises a manually annotated corpus and a set of trained models that can be used for automatic annotation (Information Extraction).

3.3.1 The OLLIE Application

OLLIE is designed as a client-server application, in order to enable training (which is CPU intensive) to be performed on the server, without interfering with the responsiveness of the client.

The client-side of OLLIE consists of a set of web-pages, with integrated Java applets, that support the interactions between the user and the system. The three main functions of the OLLIE client are: (i) support user authentication and profiles; (ii) provide ML configuration facilities; (iii) support collaborative data annotation.

Every user has a user name and a password, used to retrieve their profiles and determine which documents they can access. The profiles specify the types of annotations that are being created, and the configuration parameters for the ML algorithms used for server-side learning. These include the type of the algorithm, the specification of the feature vectors, and parameters specific to the individual learning algorithm, such as thresholds, or smoothing values.

Since OLLIE needs to support the users with the annotation process by learning in the background and suggesting annotations, it offers control over the accuracy threshold for these suggestions. The system continuously trains and evaluates a model in the background, but it only starts using it when the performance level reaches the specified threshold. This avoids annoying the users with too many wrong suggestions while ensuring that the suggestions that the system is confident about are used to pre-annotate the
data, reducing the workload of the user.

While the client side of the OLLIE application is presented as a set of web pages which can be accessed by any Java-enabled browser, the server part is based on the GATE framework. The back-end of OLLIE uses GATE to provide language processing components, services for persistent storage of user data, and application management.

Linguistic data (i.e., annotated documents and corpora) are stored in a database on the server, in order to achieve optimal performance, concurrent data access, and persistency between working sessions. The OLLIE server uses GATE’s database persistence mechanism not only to store data, but also to maintain security. All users are provided with a mechanism to authenticate themselves to the system, and they can select who else will be able to see or modify the data they store on the server.

### 3.3.2 Collaborative and IE-supported Data Annotation

OLLIE supports collaborative annotation of documents and corpora by allowing their shared, remote use and by making updates made by one client immediately available on the OLLIE server. In this way several users can share the annotation task. For example, one user can annotate a text for Organisations, then another for Locations. The documents reside on the shared server, which means that one user can see errors or questionable markup introduced by another user and initiate a discussion. Such collaborative annotation is useful not only for the purposes of training a learning system, but also in the wider context of creating and sharing language resources. The other aspect of collaboration in OLLIE – that between user and IE system suggesting annotations, is what we call here **IE-supported annotation**.

The OLLIE client provides facilities for loading documents and corpora for annotation from a URL, uploaded from a file, or created from text pasted in a form. A variety of formats including XML, HTML, email and plain text are supported. As part of this process, the user also specifies the access rights to the document or corpus, which determine whether it can be shared for viewing and collaborative annotation. The document editor
can then be used to annotate the text (see Figure 3.9). The right hand side shows the classes of annotations (as specified in the user profile) and the user selects the text to be annotated (e.g., “McCarthy”) and clicks on the desired class (e.g., Person). The new annotation is added to the document and the server is updated immediately (so the new data becomes available to the ML algorithms too). The client also provides facilities for deleting wrong annotations, which are then propagated to the server in a similar way.

The learning algorithm (which also runs as part of the server) is also notified of all added and deleted annotations, so the new training data can be taken into account. Since the data model used by the ML methods in OLLIE is based on instances (see Section 3.3.3), there is no need to wait for the entire document to be annotated before re-training the model on the new annotations. Instead, re-training is initiated as soon as new data becomes available from added and deleted annotations and the previous training phase is complete. This enables OLLIE to start providing suggestions as
soon as the desired accuracy threshold is reached. When the user is satisfied with the overall performance of the ML method, they can save the trained model for future use, including integration in IE applications independent of OLLIE.

3.3.3 Machine Learning Support

This section describes the use of Machine Learning (ML) for OLLIE. In order to support this, we created a new processing resource that adds ML support to GATE.

*Machine Learning* is a term that covers a variety of methods to do with identification of patterns in data. Most of the applications of ML deal with the *classification* problem, which comprises the distribution of a set of examples into a pre-defined set of classes. Each example is defined by a set of values (typically called features). The training stage constructs a model from a set of already classified examples; the application stage receives new examples and assigns them to the set of classes, making use of the model created during training.

There are already a large number of algorithms designed for doing Machine Learning, for which implementations are usually available. The problem we were faced with when trying to apply them to IE tasks was that they usually expect the input data to come in the form of feature vectors, while the data that IE systems use is comprised of text and annotations.

In order to bridge this data representation gap, we implemented a software library that can be used to automatically convert annotation data into ML examples, and, at the other end of the process, convert the results of the ML algorithm back into annotations. This library is exposed to GATE as a Processing Resource, and can be used to add ML-based components into language processing pipelines.

Our implementation uses a specified type of annotations as ML examples (or *instances*). The values for the features of each instance are collected from the context in which the instance annotations occur in the documents. To avoid confusion, we shall, from now on, use the term *attribute* to refer to ML
Chapter III: Portable Information Extraction: Two Approaches

features, reserving the term feature for typical GATE features associated with annotations.

Three types of attributes are supported: nominal, boolean and numeric. The nominal attributes have a specified set of permitted values, while the boolean and numeric ones have the usual definitions.

When collecting training data, all the annotations of the type specified as instances are listed, and for each of them the set of attribute values is determined. All attribute values for an instance refer to characteristics of a particular instance annotation. In order to support contextual clues, the instance annotation used to collect feature values may be either the current instance or one situated at a specified relative position.

The value for Boolean attributes is inferred from the presence or absence of a particular type of annotation overlapping the required instance. nominal and numeric attributes refer to feature values on a particular type of annotation that overlaps the instance in scope.

One of the boolean or nominal attributes is marked as the class attribute, and the values which that attribute can take are the labels for the classes to be learnt by the algorithm. Figure 3.10 illustrates some types of attributes and the values they would take in a particular example. Each attribute is defined by:

- an annotation type;
- optionally, a feature name; and
- the relative position, with regard to the current instance annotation.

The annotation type and feature name are represented a JAPE-like notation Type.feature; the relative position is indicated in brackets, with 0 indicating the current instance annotation. In the example depicted in Figure 3.10, annotations of type Token are used as ML instances.

An ML implementation has two modes of functioning: training – when the model is being built, and application – when the built model is used to classify new instances. Our implementation consists of a GATE Processing Resource that handles both the training and application phases. It is responsible for detecting all the instance annotations in a document and collecting
the attribute values for them. The data thus obtained can then be forwarded to various external implementations of ML algorithms.

During the training stage, instance annotations are used to generate training examples; during the application phase, classifications produced by the ML algorithm are used to produce annotation data. Depending on the type of the attribute that is marked as class, different actions will be performed when a classification occurs. For boolean attributes, a new annotation will be created. Nominal attributes trigger the addition of a feature value on an already existing annotation. If the required annotation is not present, it will first be created.

The execution of the ML processing resource is controlled through configuration data that selects the type of annotation to be used as instances, defines all the attributes and selects which ML algorithm will be used and
with what parameters.

The ML support library that we build does not actually perform any machine learning itself; it only handles the data conversions to and from an actual ML implementation, and it provides an interface with GATE. As such, it requires actual implementations of ML algorithms, which are integrated through a simple wrapper class.

One good source of implementations for many well-known ML algorithms is the WEKA library\(^2\) [Witten & Frank 99, Frank et al. 05]. Implementations provided there come with the added advantage that they provide a uniform API, so they are very easy to integrate into other tools.

Many of the ML algorithms in WEKA can provide a probability distribution rather than a simple classification. When available, this is converted into a confidence value, that can be used to adjust the desired balance between precision and recall.

With the support of this data conversion library, a variety of linguistic data can be used as features for training ML models. As illustrated in Figure 3.10, this can include the results of tokenisation, part-of-speech tagging, gazetteer look-ups, etc.

### 3.3.4 OLLIE Experiments and Evaluation

We performed several experiments in training the OLLIE ML algorithms to perform named entity recognition on the MUC-7 corpus [SAIC 98]. We concentrated on the recognition of ENAMEX entities, i.e., Person, Organization, and Location. The MUC-7 corpus contains 1880 Organization (46%), 1324 Location (32%), and 887 Person (22%) annotations in 100 documents. The task has two elements: recognition of the entity boundaries, and classification of the entities in the three classes. The setup and results of the experiments on each of these tasks are described below.

\(^2\)WEKA homepage: http://www.cs.waikato.ac.nz/ml/weka/
Detecting Named Entity Boundaries

We first performed some experiments to test the ability of the system to identify correctly the boundaries of named entities (comprising Organisations, Persons and Locations). Using 10-fold cross-validation on the MUC 7 corpus described above, we experimented with different machine learning algorithms and parameters (using WEKA), and using different attributes for training.

We experimented with five different algorithms, described below:

ZeroR : is a very simple algorithm that chooses the most frequently occurring class for all input examples; completely ignoring the features for these examples. It is only used here as a baseline.

OneR : is another very simple algorithm. It uses the training data to find the feature that best predicts the class of the example, and then uses only that feature during application. This one of the simplest possible learning algorithms and can be seen as a somewhat better baseline.

Naive Bayes : was chosen as a typical example of probabilistic classifier. It also has the advantage of high efficiency during both training and application.

IBK : is a particular implementation of the k-nearest neighbours (KNN) algorithm, an example of instance-based learning algorithms. The training phase is very fast, as it only consists of storing the training examples. The application phase consists of comparing the input example with the most similar ones observed in training. If the training set is very large, then the application phase can be very slow.

J48 : an implementation of the C4.5 decision trees algorithm. Decision trees have been used successfully on many applications of ML, and have the advantage that their model is humanly understandable.

The best results were achieved using the J48 algorithm with the following set of attributes:
• **For the current instance:**
  - whether it is in a gazetteer list or not
  - the name of the gazetteer list it is in
  - POS category
  - Orthography (upper case, lower case, initial upper case letter, mixture of upper and lower case)
  - Token kind (word, symbol, punctuation or number)
  - Sentence boundary

• **for position -1:**
  - whether it is in a gazetteer list or not
  - the name of the gazetteer list it is in
  - POS category
  - Orthography
  - Token kind
  - Sentence boundary

• **for position +1:**
  - whether it is in a gazetteer list or not
  - the name of the gazetteer list it is in
  - POS category
  - Orthography
  - Token kind
  - Sentence boundary

• **for position -2:**
  - POS category
  - Orthography

• **for position +2:**
  - POS category
  - Orthography

We found that using additional information, such as features on a wider window of tokens, tended to improve the recall marginally, but decreased the precision substantially, resulting in a lower F-measure, and therefore the trade-off was not worthwhile. Table 3.1 shows the number of correctly
classified instances (i.e. correctly identified boundaries) for each algorithm evaluated.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>F-measure (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZeroR</td>
<td>0.0</td>
</tr>
<tr>
<td>OneR</td>
<td>1.0</td>
</tr>
<tr>
<td>Naive Bayes</td>
<td>56.2</td>
</tr>
<tr>
<td>IBK</td>
<td>59.6</td>
</tr>
<tr>
<td>J48</td>
<td>61.2</td>
</tr>
</tbody>
</table>

Table 3.1: Performance of different algorithms on entity boundary detection task

We also tested the algorithms on a smaller news corpus (which contained around 68,000 token instances as opposed to 300,000 for the MUC7 corpus). Again, the J48 algorithm scored highest, with the K Nearest Neighbour scoring approximately 1 percentage point lower.

As expected, the two baseline algorithms (ZeroR, and OneR) do not perform particularly well. This shows that the problem being solved is not trivial. The data is greatly imbalanced, with the number of tokens that are not part of an entity being orders of magnitude greater than the ones that are. The simple algorithms used as baselines do not have the sophistication to deal with this.

We can also observe that the differences between the three fully-fledged algorithms are not great – all the results are within a 5% margin. This shows that very different ML techniques can be applied successfully to this type of problems.

**Classifying the Named Entities**

The second set of experiments was to classify the named entities identified into the three ENAMEX categories: Organisations, Persons and Locations. Using 10-fold cross-validation on the MUC 7 corpus described above, we experimented with the WEKA machine learning algorithms and parameters, and using attributes for training similar to those used for boundary detection.

The best results were achieved again with the J48 algorithm, using the
attributes from the previous experiment, without Token kind for positions \(-1\) and \(+1\), and POS category for positions \(-2\) and \(+2\). The attributes were chosen on the basis of their information gain, calculated using WEKA’s attribute selection facilities. The results are shown in Table 3.2.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>F-measure (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZeroR</td>
<td>46.0</td>
</tr>
<tr>
<td>OneR</td>
<td>85.5</td>
</tr>
<tr>
<td>Naive Bayes</td>
<td>87.3</td>
</tr>
<tr>
<td>IBK</td>
<td>88.8</td>
</tr>
<tr>
<td>J48</td>
<td>89.1</td>
</tr>
</tbody>
</table>

Table 3.2: Performance of different algorithms on entity classification task

The results indicate that this is a much simpler problem. The very simple baselines are this time reaching significant levels of performance: ZeroR correctly classifies almost half of the input examples, while OneR is less than 4% below the best result.

The different ML algorithms had different memory requirements and execution speeds. From all algorithms tested, the decision table and decision tree were the slowest (325 and 122 seconds respectively on 68,000 instances) and required most memory - up to 800MB on the big data-sets. Naive Bayes was very fast (only 0.25 seconds) with 1R following closely (0.28 seconds).

This large difference in execution time, coupled with the relatively small difference in performance between the best performing algorithm and, e.g. Naive Bayes, shows that there are practical advantages to a set-up where multiple ML algorithms are made available. In the case of interactive applications, such as the manual annotation tools in OLLIE, it may be preferable to employ a fast algorithm in order to ensure system responsiveness, even when that algorithm does not offer the best possible results. For fully-automated batch processes, the execution speed may not be of great importance, and so, slower (but better-performing) algorithms can be used.
3.4 Summary

This chapter has presented JAPE, an engine for annotation-based finite state transducers, and OLLIE, a platform for collaborative annotation backed by automatic learning.

JAPE is relevant when developing IE systems through knowledge engineering. It provides a language for describing a set of pattern-action rules, which are then grouped together into phases and grammars. An execution environment is also provided, which runs the phases in a cascaded fashion, mechanically transforming the document annotations until the required analysis is performed.

OLLIE was developed to aid when applying a Machine Learning approach for the development of IE systems. It provides an integrated on-line environment for experimenting with different ML algorithms, supports the creation of collaboratively-annotated corpora and the training of learning models in parallel.

Another set of contributions, of a more infrastructural nature, is presented in Appendix A. While each of them is of limited use individually, taken together they provide help for a broad range of problems encountered when developing IE systems.
Chapter III: Portable Information Extraction: Two Approaches
Chapter 4

Case Studies and Evaluation

This chapter evaluates the success of the work presented in this thesis. A series of case studies are introduced, which are then used to produce quantitative evaluation figures. We then reprise the set of portability dimensions mentioned in the introduction chapter and assess our success in relation with each of them.

Another, less precise, way of measuring the success of this work is to evaluate the impact and the acceptance by the language engineering community. See Appendix B for a discussion.

4.1 Case Studies

This section presents a number of case studies where the set of tools described in the previous chapters was deployed to address IE tasks. The choice of experiments is intended to illustrate the benefits along the various portability dimensions mentioned in the introduction.

4.1.1 Case Study 1: Large Scale Semantic Annotation of Patent Documents

The first case study relates to an application where IE tools and techniques were used to automatically annotate, on a very large scale, a collection of En-
Chapter IV: Case Studies and Evaluation

Figure 4.1: Reference annotations

glish patent documents from the United States Patents and Trademark Office (USPTO) and the European Patent Office (EPO), [Agatonovic et al. 08].

The main aim of the work was to create meta-data for patents in the form of annotations that are then linked with an ontology and its associated instance database, which makes it a semantic annotation task. The types of annotations created are:

Reference

Reference annotations cover text segments that are used to identify other objects. Targets of references can be other parts of the current document, such as figures, tables, examples, or patent claims. Reference annotations are also used for external entities such as literature citations or references to other patents. Figure 4.1 shows a document segment with highlighted Reference annotations of various sub-types.

Section

Patent documents have a well defined structure, with several mandated sections and some optional ones. The type of information and the language used varies between different sections – some are mainly of legal relevance while others contain technical descriptions of the invention. Because of this, it is quite useful to identify the sections of the document. In the case of this application, this is done automatically, with the results being stored as
measurements: Figure 4.2 illustrates some of the section types that are identified.

**Figure 4.2: Section annotations**

**Measurement**

The most semantically-rich annotations produced identify various types of measurements:

**Scalar Measurements** are the most usual types of measurement, comprising a scalar value and a measurement unit.

**Discrete Measurements** are measurements where the set of values is limited, for example, shirt sizes which can be S, M, L, XL, and XXL.
Interval Measurements are measurements that refer to a range of values, defined by two other measurement values.

Compound Measurements are measurements that, while referring to a single value, are described in terms of several measurements. Examples include describing a person’s height as a number of feet and a number of inches.

Figure 4.3 shows some scalar and interval measurements.

The IE application was built using the GATE framework and made use of several resources from ANNIE – the authors list the tokeniser, gazetteer and JAPE.

One distinguishing factor of this experiment is the very large scale of the data that was used:

In order to evaluate the consistency in the application’s performance on a large data set, experiments were carried out on a corpus consisting of 1.3 million USPTO (108GB) and 27 thousand EPO (780MB) documents in XML format […] 

[Agatonovic et al. 08]
4.1.2 Case Study 2: Semantic Analysis of TV News Broadcasts

The second case study looks at the RichNews application, that deployed IE techniques with the aim of automatically enriching archived news broadcasts from the BBC [Dowman et al. 05].

“The problem that the work described in this paper sought to address was that of how to improve access to the large amounts of broadcast audio and visual material produced by media organizations. Material can only be effectively accessed if meta-data describing it is available in some sort of cataloguing system. Production of such meta-data normally requires manual annotation by an archivist, a time consuming and hence costly task. This paper describes a system, Rich News, that can annotate audio and video files automatically, producing both textual descriptions and summaries, and semantic annotations that can form part of the Semantic Web.

The British Broadcasting Corporation (BBC), who created the material on which RichNews was developed, produce material for four television channels, nine network radio stations, and numerous local radio stations. Annotation of this material is an expensive and labor-intensive task. For example, it takes a BBC archivist almost seven hours to catalog Newsnight, a fifty minute daily news broadcast, in detail. Because of the high cost of cataloging, 90% of the BBC’s output is annotated only at a very basic level, making it difficult to re-use it after its initial broadcast. Furthermore, because of the time it takes for cataloging to be completed, there is a delay before the material is available, which can be a problem in areas such as news and current affairs, when the material is most likely to be useful immediately after it is broadcast.”

[Dowman et al. 05]

The approach taken was one of multi-source Information Extraction; the input data comprises automatic speech recognition transcripts of audio-visual
broadcasts. Due to the various recording conditions (different speakers, in-
studio vs. outdoors conditions), there are a large number of recognition
errors. Word error rate was evaluated for different transcriptions, and it
was found to vary between 30% and 90%, depending on various factors (ex-
traneous noises, speaker accent, recording quality, etc.)

In order to address the problem of recognition errors, a two stage approach
was chosen:

1. First, the speech transcripts are used to split the whole broadcast
into segments, intended to represent individual stories. For each such
segment, key-phrases are identified using statistical methods based on
$TF\cdot IDF$ [Salton & McGill 83].

2. The key-phrases found during the first step, together with the date
of the original broadcast, are used to locate web pages (mainly on
BBC’s own web site) that present the same news story as the segment
being analysed. The best matching candidate pages are chosen based
on lexical similarity and are then analysed with a classic IE system,
based on ANNIE.

Because the text in the web-pages is error free, grammatically correct, with
good capitalisation and punctuation, the quality of the results is expected
to be superior to directly analysing the speech transcripts.

4.1.3 Case Study 3: Ontology Authoring with Controlled
Language Information Extraction

This next case study refers to an application that uses IE techniques to parse
a controlled language used to author ontologies [Tablan et al. 06a].

The motivation for this work stems from the complexity of current tools
for ontology editing, which are difficult to use because they need to
be compliant with elaborate ontology representation languages such as
OWL [Dean et al. 04] or RDF-S [Lassila & Swick 99].

In many cases where ontologies are used within NLP systems, only a few of
the features supported by languages like OWL are actually used. In most
cases, the requirements only include representing a taxonomy of classes, be-
ing able to define properties that apply to class members, and the ability
to describe new instances for the classes and the property values for them.
However, in order to insure compatibility, the format used for data inter-
change needs to comply with the standards. This creates a situation where
relatively simple information, i.e. that people would find easy to put into
words, needs to be represented using an advanced mechanism that requires
a deep understanding of the intricacies of the representation formalism.

This motivated the authors to build a tool that enables users to create and
edit ontologies simply by using a restricted version of the English language.
The controlled language used is based on an open vocabulary and a re-
stricted set of grammatical constructs. Sentences written in this language
unambiguously map to semantic constructs that can be represented formally
as an ontology.

The possible types of actions are: definition of new classes, creation of
hierarchies between classes, definition of object and data-type properties,
creation of instances, and setting of property values for instances. The full
set of contracts permitted in the controlled language is included in Table 4.1.

<table>
<thead>
<tr>
<th>Sentence Pattern</th>
<th>Usage</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>There are &lt;class&gt;.</td>
<td>Declares new classes</td>
<td>There are publications.</td>
</tr>
</tbody>
</table>
| <sub-class> is a type of <super-class>. | Declares a new class as a subclass of an exist-
ing class. | Book is a type of publication.                    |
| <sub-class>, ... <sub-class> are a type of <super-class>. | Declares several new classes as subclasses of an existing class. | Books and articles are a type of publication. |
| <class> (can) have <class>. | Declares a new property linking two classes. | Publications have authors.                        |
| <class> (can) have textual <property name>. | Declares a new datatype property of type string. | Publications have textual title.                  |
Chapter IV: Case Studies and Evaluation

<table>
<thead>
<tr>
<th>&lt;instance name&gt; has &lt;instance name&gt;.</th>
<th>Gives a value for a property of an instance. The name of the property is obtained from the domain and range restrictions of the known properties.</th>
<th>Book1 has John Smith.</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;instance name&gt; has property &lt;property name&gt; with value &lt;instance name&gt;.</td>
<td>Gives a value for a specified property of an instance. This variant can be used when there are several properties that might apply for the given instance types.</td>
<td>Book1 has author with value John Smith.</td>
</tr>
</tbody>
</table>

Table 4.1: Controlled language constructs

One advantage of this approach is that it requires essentially no training; there are no complicated user interfaces to be learnt; there are no complex formalisms to be understood. The user can simply start from a simple example which shows all of the types of utterances accepted by the system, and continue the ontology authoring work by re-using and modifying the provided examples.

The application that performs the parsing of the controlled language is implemented as a pipeline of GATE processing resources that reuses some of the ANNIE components. A schematic of the implementation is shown in Figure 4.4.

An example of controlled language text and the resulting ontology is presented in Figure 4.5.

More recent work, in a similar vein, deals with using natural language for querying ontologies and is presented in [Tablan et al. 08].
4.1.4 Case Study 4: Linguistic Analysis of Sumerian Literary Works

The final case study refers to work done to enrich the Electronic Text Corpus of Sumerian Literature (ETCSL)\(^1\) based at the Oriental Institute at the University of Oxford, [Tablan et al. 06b].

The Sumerian language of ancient Sumer is a long-extinct language documented throughout the ancient Middle East, in particular in the south of modern Iraq, from at least the 4\(^{th}\) Millennium BCE. It is arguably the first language for which we have written evidence, the rival candidate being ancient Egyptian. Sumerian was replaced by Akkadian as a spoken language around 2000 BCE, but continued to be used as a sacred, ceremonial and scientific language in Mesopotamia until about 1 CE.

The Electronic Text Corpus of Sumerian Literature (ETCSL), based at the University of Oxford, aims to make accessible on the web over 350 literary works composed during the late third and early second millennia BCE. The corpus comprises Sumerian texts in transliteration, English prose translations and bibliographical information for each composition.

[Tablan et al. 06b]

The work described here deals with the creation of linguistic analysis and corpus search tools for Sumerian, as part of the development of the ETCSL.

\(^1\)http://www-etcsl.orient.ox.ac.uk/
There are publications and authors. Papers, articles and books are a type of publication. Publications have authors. Publications have textual titles. Paper 1 is a paper. John Smith is an author. Paper 1 has author with value John Smith. Paper 1 has title with value "A Paper About Something".

Figure 4.5: Example of input text and resulting ontology

This was designed to enable Sumerian specialists to analyse the texts on-line and electronically and to further knowledge about the language.

The main aim of the work was to create a set of tools for performing automatic morphological analysis of Sumerian. This entails identifying the part of speech for each word in the corpus (technically, this only involves nouns and verbs, which are the only categories that are inflected), separating the lemma part from the clitics and assigning a morphological function to each of the clitics.

While the task performed was not strictly speaking one of Information Extraction, we feel this work is relevant as the approach and methodology used are inspired from IE research and because it provides a good example of applying the same tools in a new domain.

In order to perform this analysis, a model of Sumerian morphology defined by a team of Sumerologists, was used. It consists of noun and verb templates comprising a lemma plus a number of morphological slots that could be filled.
The nouns have a lemma and up to six suffix slots while the verbs have up to twelve prefix slots, a lemma and two suffix slots. For each slot there is a known list of morphemes that can fill it and a set of restrictions encoding dependencies between the slots, such as agreement in gender. The lists of candidate slot fillers have non-null intersections – the same morpheme can appear in several lists, though usually with different functions.

As there is no Unicode range for Sumerian cuneiform, the input texts used were represented as Roman transliterations.

The main stages of the process are:

**Tokenisation:** splits the input text into syllables while also identifying special text elements such as determiners and markers for damaged regions in the original clay tablet. This is performed using the ANNIE Tokeniser with a customised rule set.

**Input normalisation:** deals with some surface phenomena typical of the Sumerian language, such as reduplication (where syllable-final letters are repeated at the start of the next syllable) or assimilation, where letters are deleted. The approach taken is to make explicit the ambiguity by generating all possible normalised interpretations for each particular text fragment. It falls to the next stages to delete the incorrect variants. This is implemented through a combination of ANNIE gazetteers and JAPE grammars.

**Slot fillers look-up:** identifies syllables in the input that are candidate fillers for morphological slots. The implementation is based on ANNIE gazetteers.

**Non-inflected words lookup:** identifies words that are not inflected by looking them up in a predefined list. This step is implemented using an ANNIE gazetteer.

**Morphological analysis:** identifies nouns and verbs and generates structure information by labelling the lemma and all the other constituents. This stage takes into consideration all possible interpretations based on the ambiguous input, it filters them by applying morphological constraints, and it then applies a set of heuristics to choose the most likely
variant if there are still multiple candidate interpretations remaining. This is implemented through a cascade of JAPE grammars.

The results obtained are presented in Table 4.2.

<table>
<thead>
<tr>
<th>Type</th>
<th>P</th>
<th>R</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>noun recognition</td>
<td>59%</td>
<td>84%</td>
<td>69%</td>
</tr>
<tr>
<td>verb recognition</td>
<td>65%</td>
<td>67%</td>
<td>66%</td>
</tr>
<tr>
<td>morphological analysis</td>
<td>52%</td>
<td>73%</td>
<td>61%</td>
</tr>
</tbody>
</table>

Table 4.2: Evaluation of POS recognition and morphological analysis

4.2 Quantitative Evaluation of Software and Resources Reuse

For each case study described in section 4.1, we shall evaluate the support provided by the work described in this thesis. This will be done by measuring the quantity of additional software and linguistic resources that needed to be created compared to the pre-existing ones that were simply reused.

<table>
<thead>
<tr>
<th>Software code (lines)</th>
<th>Linguistic resources (lines)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tokenisers</td>
<td>1,656</td>
</tr>
<tr>
<td>Gazetteer</td>
<td>1,130</td>
</tr>
<tr>
<td>Sentence splitters</td>
<td>700</td>
</tr>
<tr>
<td>POS Tagger</td>
<td>405</td>
</tr>
<tr>
<td>JAPE</td>
<td>3,380</td>
</tr>
<tr>
<td>Controllers, etc.</td>
<td>1,455</td>
</tr>
<tr>
<td><strong>Total:</strong></td>
<td><strong>8,726</strong></td>
</tr>
</tbody>
</table>

Table 4.3: Lines of software code and linguistic resources for various components

In order to support these quantitative measurements, we started by estimating the amount of software and linguistic resources included with ANNIE; the resulting figures are presented in Table 4.3. This includes only the software
elements required to execute the language engineering process, i.e. processing resources and application controllers. This can be seen as the runtime element of ANNIE. It is likely that other elements of the work described here, especially the infrastructural support presented in Appendix A and the development environment, were also useful. However we decided to not count them as reused software because it is difficult to measure their direct contribution and because it it theoretically possible, though more difficult, to develop an IE application without using them.

We were also careful not to include the work of other GATE contributors, for instance in the cases where processing resources are making use of external libraries, we only counted the source code created to wrap that functionality and not the source code of the original library. It should be noted at this point that, due to the open-source nature of the GATE project (which incorporates the work presented in this thesis), some other contributors may have submitted changes to the implementation of some of the software modules. However, all of these software components were originally designed, developed and are maintained by the author of this thesis; any externally-contributed changes are minor in nature.

The set of linguistic resources included with ANNIE, i.e. grammars, lexicons, rule-sets, etc., have been collected from a variety of sources, including publicly available databases of names, the results of other projects, through automatic learning, or were created by various members of the GATE community. While not directly authored by us, we feel they are relevant to this evaluation because:

1. They were created specifically for the software tools developed here. It is reasonable to assume that without this work, the set of resources would also not exist.

2. They are publicly available, so any person using the software can also benefit from the use of these resources.

For each case study presented in Section 4.1, we used the same methodology to measure:

**Reused software:** the quantity of software developed by us that was directly reused in the implementation;
Chapter IV: Case Studies and Evaluation

**External software:** the quantity of software from other sources (including other GATE components outside the scope of this thesis) that was reused;

**New software:** the quantity of new software that needed to be created;

**Reused resources:** the amount of ANNIE resources that were reused;

**External resources:** the quantity of other external resources;

**New resources:** the amount of newly developed resources.

In all cases, the quantities are measured in terms of number of lines. The values obtained for the four case studies have been tabulated and are presented in Table 4.4.

<table>
<thead>
<tr>
<th>Case Study 1 (Patents)</th>
<th>Software</th>
<th>Resources</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Reused</td>
<td>External</td>
</tr>
<tr>
<td>Case Study 1 (Patents)</td>
<td>7,596</td>
<td>5,271</td>
</tr>
<tr>
<td>Case Study 2 (TV News)</td>
<td>8,726</td>
<td>11,230</td>
</tr>
<tr>
<td>Case Study 3 (Controlled Language)</td>
<td>8,726</td>
<td>3,763</td>
</tr>
<tr>
<td>Case Study 4 (Sumerian)</td>
<td>6,166</td>
<td>466</td>
</tr>
</tbody>
</table>

**Table 4.4: Reuse of software and resources**

The same information is shown graphically in the form of pie-charts in Table 4.5, below. This gives a better view with regards to the proportions between the elements.
Case Study 1 (Patents)

<table>
<thead>
<tr>
<th>Software</th>
<th>Resources</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reused</td>
<td>55.29%</td>
</tr>
<tr>
<td>External</td>
<td>38.37%</td>
</tr>
<tr>
<td>New</td>
<td>6.35%</td>
</tr>
<tr>
<td>Reused</td>
<td>31.72%</td>
</tr>
<tr>
<td>External</td>
<td>11.93%</td>
</tr>
<tr>
<td>New</td>
<td>56.35%</td>
</tr>
</tbody>
</table>

Case Study 2 (TV News)

<table>
<thead>
<tr>
<th>Software</th>
<th>Resources</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reused</td>
<td>31.74%</td>
</tr>
<tr>
<td>External</td>
<td>40.85%</td>
</tr>
<tr>
<td>New</td>
<td>27.41%</td>
</tr>
<tr>
<td>Reused</td>
<td>1.39%</td>
</tr>
<tr>
<td>External</td>
<td>9.12%</td>
</tr>
<tr>
<td>New</td>
<td>89.49%</td>
</tr>
</tbody>
</table>

Case Study 3 (Controlled Language)

<table>
<thead>
<tr>
<th>Software</th>
<th>Resources</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reused</td>
<td>9.40%</td>
</tr>
<tr>
<td>External</td>
<td>27.30%</td>
</tr>
<tr>
<td>New</td>
<td>63.30%</td>
</tr>
<tr>
<td>Reused</td>
<td>2.61%</td>
</tr>
<tr>
<td>External</td>
<td>26.56%</td>
</tr>
<tr>
<td>New</td>
<td>70.63%</td>
</tr>
</tbody>
</table>
Chapter IV: Case Studies and Evaluation

4.3 Portability Dimensions

In the introduction of this thesis, we referred to a set of portability dimensions that we aimed to address. Our aim was to provide a tool set that is applicable for building IE systems throughout the whole space defined by these dimensions. Let us look at them again, one by one, and assess the success of this effort.

4.3.1 Task

*Information Extraction* is an umbrella term that covers a whole family of problems. Different tasks involve different combinations of IE data types of varying complexity.

Looking back at the case studies presented at the start of this chapter, we can justify the claim that they cover a range of widely differing tasks:

Case Studies 1 and 2 deal with identifying domain entities, e.g. *Measurement* and *Reference* annotations in the first case, *Person* and
Location names in the latter.

They also identify domain relations, such as in the case of Interval Measurements in the first case study, which are essentially represented as a relationship between two other measurements.

The first case study also includes the identification of document sections, which can sometimes span whole pages. This is a task quite different from entity discovery, which relies mainly on local analysis and results in annotations with relatively short spans.

The application Case Study 3 works by identifying prescribed syntactic structures with an open vocabulary.

Finally, Case Study 4 fills in morphological templates whilst attempting to satisfy a complex set of restrictions.

4.3.2 Genre and Style

The language involved in the input of IE systems can vary from formal and precise (e.g. news articles) through to informal and badly spelled (e.g. emails).

The text types that our case studies consume vary across a range of genres and styles, including:

- patent documents which include legal language and technical descriptions;
- the output of automatic speech recognition systems, which is lacking capitalisation, and presents numerous word errors;
- journalistic text in the form of web pages presenting news stories;
- an artificial controlled language, and
- transliterated Sumerian.
4.3.3 Domain

The domain of the IE application, i.e. the subject matter of the texts to be analysed, is one of the most relevant dimensions of portability. Adapting a system from one domain to another is a very frequently occurring requirement in many organisations that have invested significant resources in deploying IE technology.

Our case studies provide a good example of such portability: they apply to domains as varied as Intellectual Property, news, ontology engineering and scholarly study of ancient languages.

Despite this wide range of applications, significant portions of each of the systems were constructed from the selection of tools we developed. These components were used unchanged and, after also taking into consideration software obtained from other external sources, the amount of new source code necessary was below 10% in three out of four cases, with the remaining one only reaching 27.41%, i.e. still less than one third.

This demonstrates a high degree of domain portability and indicates a significant reduction of the effort required to port an IE system to a new domain, compared to developing one from scratch.

4.3.4 Language

Another frequently occurring case of porting IE systems is adaptation to new languages.

Providing portability of IE systems to new languages in the general case is probably an intractable problem due to the high variability in morphology, writing system, syntax, etc. between various languages. However, that does not mean that steps cannot be taken to make the task more approachable. In the course of this thesis we have touched on several issues that are relevant in this context:

**Infrastructural support** presented in Appendix A provides pluggability that allows components of an IE system to be swapped with other tools. This is particularly useful for basic analysis tools, such as tokenisers.
or part of speech taggers, which may already be available for the new language.

Separation between software and resources is also important: in many cases, the same software tools can be used for different languages, the only changes being required in the linguistic resources. Our case study 4 is a good example of this: developing the system for Sumerian required completely new resources, while almost 90% of the software was reused.

Multilinguality support presented in Section A.5.1, such as being able to view and edit text using different writing systems can also help when working with new languages.

One initiative dedicated to assessing the portability of various NLP tasks to new languages was the 2003 TIDES Surprise Language Exercises [Oard 03]. This included, amongst others, the task of entity tagging, which is another name for named entity recognition. As part of this exercise, [Maynard et al. 03] has shown success in porting the ANNIE system to Cebuano and to Hindi, each taking about one week of effort.

### 4.3.5 Scale

IE applications requirements vary from the small scale to the size of the web itself, and processing requirements vary from overnight to near real time.

In order to support such requirements, the algorithms used should be efficient and the implementations as reliable and robust as possible. Most of the tools presented in this thesis employ finite state machinery and regular expressions, which are known as a class of very efficient methods for dealing with text. As shown in Section 3.2.5, JAPE, one of the main elements of this effort, has a linear time complexity in typical applications.

As a scalability test, Case Study 1, above, has shown that our tools can be successfully applied to a very large collection of documents.
4.3.6 End-user Type

The types of users that require deployments of IE systems vary in terms of language engineering expertise. As IE technology leaves the research laboratories and is deployed into real-life applications, some of its users may have a sophisticated knowledge of computing and/or linguistics, whereas others may have almost none.

This issue is addressed by our provision of supporting tools for approaches that rely both on language engineering (see Section 3.2) and on machine learning (see Chapter 3.3). Experienced language engineers can use the former, while users that have little knowledge of how IE systems are built can opt to employ the latter.

4.4 Summary

In this chapter we have shown, through the lens of four different application scenarios, how the tools and methodology described in this thesis help in improving the portability of the IE systems. We have shown that the effort for creating a new system (or re-purposing an existing one) is reduced as a result of our contribution.

We should note here that our intention was not to create a generic IE system that is suitable for any domain, language, and application. We believe that is still far beyond the current state-of-the-art. What we are offering instead is a set of tools and formalisms that reduce the number of adaptations needed when travelling along the various portability dimensions.
Chapter 5

Conclusions and Future Work

In this thesis we have presented work done in support of portability for Information Extraction systems. We took a multi-dimensional approach aimed at addressing the various aspects of portability, ranging from different input types, different IE tasks, and different methodological approaches to performing IE.

5.1 The Knowledge Engineering Approach

In support of the Knowledge Engineering approach, Section 3.2 of Chapter 3 was dedicated to JAPE – an engine for building annotation transducers based on pattern-action rules.

The paradigm of annotation transducing provides a powerful and flexible way of implementing language engineering processes by specifying a set of rules that use annotation-based patterns as input and produce other annotations as output. Several JAPE transducers can be chained together into a cascade where each step has access to all the annotations produced by the previous ones.

JAPE was developed as a formalism that allows linguists and software engineers to express their linguistic knowledge as a set of rules that can be
followed by a machine in order to perform linguistic analysis tasks. It is somewhat analogous to a programming language in the sense that it is intended to be easily authored by humans and understood and executed by computers. Being dedicated to language processing tasks, the data structures that it uses comprise documents and annotations, while the actions performed consist of manipulating such annotations.

The language used to define the annotation patterns is inspired by the regular expressions formalism. Simple pattern elements match single annotations by specifying restrictions based on annotation types and features; complex patterns are then built from these elements by combining them via disjunction, juxtaposition, or Kleene operators. The input used by the matching process is the graph structure formed by the annotations covering a GATE document. To deal with the issues caused by the input being a graph and not a simple sequence, various matching styles are defined that can be used to affect the way the annotation patterns match annotations on the input document. Paying tribute to the field of formal languages, a set of rules for a transducer is called a JAPE grammar.

The main aim behind the development of JAPE was to support the developers of IE systems in creating rule sets for identifying linguistic patterns indicative of the information types that need to be extracted. These rules identify relevant text snippets by modelling their context or internal structure, and can make use of all the annotations produced by basic linguistic analysis modules (such as tokenisers, part-of-speech taggers, etc.)

Besides its use for extraction rules, JAPE is also convenient as a generic tool for language engineering tasks. It can be used for rapid prototyping of modules, and trying out of ideas. When the optimal solution is found, the JAPE-based prototype can be used as a specification for a dedicated module implemented as a separate piece of software. JAPE transducers also provide a simple way of performing basic annotation manipulation tasks, such as format transformations, gluing together of other components, simple post-processing of the output of other modules, etc.

The JAPE implementation distributed with GATE has proved very popular with the community of GATE users, generating about one third of all the email traffic on the dedicated mailing list (see Appendix B).
5.2 The Machine Learning Approach

Building IE systems using the learning approach involves the creation of a large corpus of training data which is then used to train a Machine Learning (ML) algorithm. Machine learning works by analysing a large number of training examples for a particular phenomenon, building a model that represents the training data, and then using that model to analyse new examples. Each example is defined by the values of a set of features. For example, in the case of people, such features could be the name, height, or the age of the person.

The most typical task for ML systems is one of classification, where each training example has an associated label, also known as a class or category. Once trained, an ML system is then used to associate new examples with labels from the pre-defined set. This process is known as classification or categorisation.

Machine Learning is a very active research area, with many algorithms having already been developed, and new ones being constantly created, some of which are especially developed for language processing tasks. Because of this, our aim was not to create yet another algorithm to be used for Information Extraction; we chose instead to build the infrastructure required to expose IE tasks as classification problems. This allows the use of any of the many existing algorithms for solving IE tasks, and encourages experimentation with various different ML implementations until the most suited one is found.

Our work in this direction is presented in Section 3.3 of Chapter 3 and is centred around the OLLIE application, which provides an integrated on-line environment for experimenting with different ML algorithms. The main contribution was the development of a software library that converts annotations to feature sets, as used by ML algorithms (see section 3.3.3). This forms the base of the ML support in GATE and can be used for many types of language processing tasks, including information extraction.

Besides providing a generic way of converting IE tasks into classification problems, OLLIE is also an on-line annotation environment, which allows users to share and collaboratively annotate distributed corpora. During the
Chapter V: Conclusions and Future Work

annotation process, ML is used in the background to provide suggestions for new annotations, which should reduce the total effort required for the construction of the annotated data. While this approach is similar to other adaptive IE environments and collaborative annotation tools, OLLIE is more flexible in that it allows users to choose which ML algorithm they want to use and to comparatively evaluate different approaches. This makes it more valuable as a platform for experimentation.

5.3 Knowledge Engineering or Machine Learning?

The question of which approach is the right one is often asked when an IE system is being developed. This section includes a discussion covering some of the factors that inform this decision.

Seen from a purely functional point of view there is no clear winner: both approaches have been used successfully for IE tasks and with similar levels of performance. The question then becomes one of minimising the cost associated with building a new IE system or with adapting an existing one to a new task. When viewed from this perspective, the most appropriate approach to take is the one that is most cost-effective. When comparing the costs of the two approaches, it is usually the case that deploying a Knowledge Engineering (KE) solution requires a team of highly skilled, and thus expensive, language engineers. On the other hand, the Machine Learning (ML) approach involves creating large amounts of training data through manual annotation. Typically, the manual annotation process takes more person-hours of effort but the annotators are usually less skilled and less expensive than the language engineers. The final choice should then be made based on how these two costs compare.

In each particular case, there are a range of factors that have a bearing, some of them external to the specifics of the actual technology. We discuss some of these next.

Availability of annotated resources
There are scenarios where there already exists a corpus of documents that include some type of annotation.
One example is the case of content providers, such as broadcasting organisations, where high-value content is already manually annotated in order to support indexing and access. This annotated data can then be used to build an IE system that would produce similar annotation to the rest of the materials in the archive, for which the manual annotation process is currently too expensive to deploy.

Another case is that of social web sites, where many entries already have tags associated with them. These tags are created either by the authors or by other members of the community and can then be used to train an IE system that would be capable to annotate new, untagged, data.

In either of these cases, the legacy annotated data can be used for training, so the ML approach is likely to be more advantageous than the Knowledge Engineering one.

**Availability of annotators**

Alternatively, if pre-existing annotated resources are not available, the question becomes *how cheaply can they be created?* If a group of low-cost annotators are available (e.g. volunteers), then the necessary training data can be created cheaply and the ML approach would again be preferred.

**The application domain**

There are domains, such as Intellectual Property Law, or fields of scientific research, where the type of content to be annotated requires highly specialised experts. In these situations, the annotators could be significantly more expensive than the language engineers, so ML would not be cost-effective.

The solution is then to deploy the KE approach, where domain experts spend a short amount of time being interviewed by the language engineers, which then go on to construct the IE system.

**Complexity of the extraction task**

Another factor that is relevant when deciding which approach to choose, is the complexity of the extraction task. In some cases the frequency of training examples can be very low and data sparseness becomes an issue: obtaining a sufficient number of training examples may require the manual annotation of a very large number of documents. This can happen, for instance, when
trying to identify complex relationships between a large set of domain entity types.

In these situations, the KE approach can prove more effective. The human engineers make use of their external knowledge and linguistic intuition when hand-crafting rules, and they are better than ML algorithms at generalising from a small set of examples.

Before we end this section we should note that the distinction made between approaches based on Knowledge Engineering and those using Machine Learning is not always clear-cut. These two approaches can easily coexist within the same system:

- Different modules of the same system can use different approaches. For instance many KE systems employ part-of-speech taggers (or syntax parsers), which are frequently constructed using some ML method.

- Different sub-tasks of the IE application can be approached through different methods. While using the KE approach for identifying entity relationships (perhaps due to data sparseness), the same system could employ an ML solution for finding domain entities (where the training examples are more frequent).

As we have seen, there is not hard rule regarding which approach is best. There are a large number of factors that influence this decision, some of which are more to do with organisational issues and cost/benefit analyses then with the technology itself. The only real way to provide IE portability is then to offer support for both and give the IE developer the option to choose the most appropriate solution in each particular case.

### 5.4 Infrastructural Support

Appendix A introduces a set of tools to provide functionality that is generally useful when developing IE systems. This includes software infrastructure for building language engineering modules and chaining them into processing pipelines, tools for supporting manual annotation, tools for performance evaluation, a set of ready-built basic language analysis modules, and tools
for dealing with semantic meta-data in the form of ontologies. The provision of these tools is relevant to the issue of systems portability in IE for a variety of reasons, which are explained below.

Providing a robust software infrastructure for language processing removes a significant amount of the effort required in designing and implementing any NLP system. Given this support, the team of software and language engineers can concentrate on solving the issues directly relating to the IE task that they are aiming to address, without having to concern themselves with basic software engineering issues such as data representation, input/output from various file formats, workflow control, data visualisation, etc.

Manually annotated data is used during various stages of the development process, starting with the initial step of refining the problem definition, ending with the performance evaluation of the final system, and during the many develop-deploy-test cycles in between. Because of this, the manual annotation of documents is a significant element of the effort spent. The graphical interface tools we provided to support manual annotation were designed to make this process as efficient as possible, minimising the number of human-computer interactions required, and the overall time spent for this activity.

The provision of tools for automatic performance evaluation is not only helping in assessing the quality of the final system but it is also used during the development process to pinpoint particular problems such as types of documents on which the system is not performing well enough, analysis components that make systematic errors, or areas of the system that require attention.

Finally, the availability of tools to perform basic language analysis tasks, such as tokenisation, sentence splitting, or part-of-speech tagging, removes some of the drudgery associated with building a new IE system. Any such system will require some of these tools, regardless of varying factors like the application domain, or input document type.

In conclusion, the infrastructural support described in Appendix A has as its main aim the reduction of the effort required in building IE systems. This allows the developers to concentrate on the specific issues related to their
Chapter V: Conclusions and Future Work

particular application set-up, with faster develop-deploy-test cycles, leading to a more agile development process.

5.5 Future Work

The work presented in this thesis is about providing a set of software tools that support the portability of IE systems by allowing rapid customisation to new problem settings. While software provision is an important aspect, there are other elements that can contribute to reducing the costs associated with IE portability. Of these, probably the most relevant centres around the issues of know-how and methodology.

Experienced teams of language engineers can be very efficient in porting IE systems, and a contributing factor to this efficiency is the availability of tools and resources as presented in this thesis. However, another important factor is using the right methodology, that is usually arrived at through experience. For example, such a methodology could include, in a typical case:

1. Textually describe the problem to be solved. This normally includes the list of information types that need to be extracted, with examples.

2. Decide how to model this problem in terms of annotations – which annotation types to use, which feature names, which pre-defined sets of values.

3. Produce some manually annotated data:
   
   (a) Produce an annotation guide, describing how manual annotation should be performed.
   
   (b) Run an experiment of parallel annotation, where several annotators mark-up the same texts.
   
   (c) Evaluate the inter-annotator agreement; if differences between annotators are too great, refine the annotation guide to better explain the task and repeat the parallel annotation experiment.
   
   (d) At the end of this process a gold standard corpus has been produced, that contains authoritative manually-annotated data.
4. Divide the gold standard corpus into training and evaluation sections.

5. Develop the IE system, using the training gold standard corpus as the task specification.

   (a) At regular intervals evaluate the performance of the system against the test section of the gold standard corpus.
   (b) Identify problem areas and develop them further.
   (c) Stop when the required level of performance is reached.

Another aspect of methodology refers to how to divide the work in the most cost effective manner. Looking at the kind of tasks that are part of constructing (or adapting) an IE system, it seems possible that several different contributor profiles can be defined, based on the amount of expertise required to carry out certain tasks. A possible set of such profiles can be:

**Language Engineers** are the ones that manage the process of constructing the IE system, and perform the skilled language engineering tasks, such as creating rule sets. Due to their high level of expertise, they can have high associated costs.

**Domain Experts** are involved in creating the task specification, as a textual description. They work together with the **language engineers** to refine this specification. Due to their high level of domain expertise, they also can have very high associated costs.

**Annotators** are people who are capable of manually annotating text according to a textual guide. They do not need to have language processing expertise, but they may require some domain knowledge. Depending on the particular setting, they can have very low associated costs (e.g. volunteers).

Other profiles could also be defined. For example, one could consider a role for team members who write the manual annotation guide (based on the task specification), manage the manual annotation experiments, and perform the inter-annotator agreement measurements. This role would probably fit a less experienced language engineer and would be placed, in terms of cost, between a language engineer and an annotator.
In this case, the methodological issue becomes one of maximising the return from the work, while minimising the costs, and without reducing the efficiency of the whole process.

Codifying such pre-existing expertise into clearly defined rules and methods can be very useful for training new team members or exporting knowledge to new groups. However, this is not a trivial task and we intend to study this problem further.
Appendix A

Infrastructural Support for Information Extraction

Regardless of whether knowledge engineering or machine learning are used, the task of building an IE system can greatly benefit from infrastructural support which helps with the basic language engineering tasks.

This appendix gives an overview of the types of infrastructural support that can be provided for the development of Information Extraction systems. The actual tools that are described here have been implemented as part of the GATE framework (version 2 and later) and were subsequently publicly distributed with it. The thinking behind GATE is to isolate those parts of language processing systems that are infrastructural and provide reusable implementations of them that can be deployed in multiple contexts. As such, it provided an excellent platform for all the implementation work.

A.1 GATE

In order to contextualise the work described in the rest of this thesis, we need to include here a short overview of GATE – an architecture, framework and development environment for Language Engineering. GATE has been in development at the University of Sheffield since 1995, with the first version being released in 1996 [Cunningham et al. 96, Gaizauskas et al. 96]. This was followed by version 2.0 in 2002 [Cunningham et al. 02], 2.1 and 2.2 in

As an architecture, GATE defines the organisation of an LE system and the assignment of responsibilities to different components, and ensures that the component interactions satisfy the system requirements. As a framework, it provides a reusable design for an LE software system and a set of pre-fabricated software building blocks that language engineers can use, extend and customise for their specific needs. As a development environment, it helps its users to minimise the time they spend building new LE systems or modifying existing ones.

Architectural considerations suggest that the elements of software systems that process natural language can usefully be broken down into various types of component, known as resources. Components are reusable software chunks with well-defined interfaces, and are a popular architectural paradigm, used in Sun’s Java Beans and Microsoft’s .Net, for example. GATE components are specialised types of Java Bean, and come in three flavours:

- **Language Resources (LRs)** represent entities such as lexicons, corpora or ontologies, that are mainly defined by the data they contain;

- **Processing Resources (PRs)** represent entities that are primarily algorithmic, such as tokenisers, taggers, or parsers;

- **Visual Resources (VRs)** represent visualisation and editing components that participate in GUIs.

Collectively, the set of resources integrated with GATE is known as CREOLE: a Collection of REusable Objects for Language Engineering. All the resources are packaged as Java Archive (or ‘jar’) files, plus some XML configuration data.

### A.1.1 A Homogeneous Data Model

Textual data come in many shapes and forms in terms of character encodings, file formats, mark-up standards, etc. This can be a cause of problems for
the implementation of language processing systems if all components need to be capable of dealing with all possible types of input.

Infrastructural support can help by providing a unified inter-operable data model which is used by all language processing components and to which various types of input are translated. This is exactly what GATE does through the set of Language Resources (LRs) that it defines and through the automatic tools that perform conversions to and from this unified model.

Concepts like corpora, documents and annotations are standard in the field of computational linguistics: they represent the types of data all NLP systems have to deal with. They also constitute the building blocks of the GATE data model.

A Language Resource is an entity that stores some form of linguistic data. There are two basic types that are predefined in GATE: the document and the corpus.

A document is a unit of linguistic data that cannot be broken apart into smaller pieces without loss of information. In GATE, a document consists of some content (a piece of text) and one or more layers of annotation.

An annotation, the basic unit in an annotation layer, is a form of metadata attached to a particular section of document content. The connection between the annotation and the content it refers to is made by means of two pointers that represent the start and end locations of the covered content. An annotation must also have a type (or a name) which is used to create classes of similar annotations, usually linked together by their semantics.

An annotation layer is organised as a Directed Acyclic Graph (DAG) on which the nodes are particular locations (anchors) in the document content and the arcs are made out of annotations reaching from the location indicated by the start node to the one pointed to by the end node (see Figure A.1 for an illustration). Because of the graph metaphor, the annotation layers are also called annotation graphs. In terms of Java objects, the annotation layers are represented using the Set paradigm as defined by the collections library and they are hence named annotation sets. The terms annotation layer, graph and set are interchangeable and refer to the same concept when used in this thesis.
All documents must have at least one annotation layer, called the default annotation set, although this can be empty. When required, several additional annotation layers can be added. These new annotation layers must have unique names to allow for distinguishing between them.

Putting together one or more documents results in a **corpus**. Although semantically a corpus is defined as a collection of documents, GATE uses lists (which are *ordered* collections) to define corpora. This does not change the model since, if document order is not important, it can be ignored; but it does help when, for practical reasons, the order of the documents in a corpus needs to be controlled.\(^1\) Putting together a set of documents as a corpus allows them to be treated as a unit by e.g. processing them through the same chain of processing resources or saving them using the same persistent storage mechanism.

GATE uses a unified model for describing the meta-data attached to data containers, such as corpora, documents and annotations, which is based on a **feature map**. The idea of a *Feature Map* (or *Feature Structure*) is well known in Artificial Intelligence. A Feature Map is used to describe attributes (features) of particular objects and consists of a set of `<feature name, feature value>` pairs. The feature names are strings and the feature values can be any Java object. The feature maps use the standard Map interface, part of the standard Java Collections library, making them inter-operable with other Java applications and libraries. Every type of GATE resource and all the annotations have associated feature maps.

\(^1\)E.g. for ensuring a chronological processing of documents originating from a news-feed.
A.2  Work-flow Control

A.2.1 Language Analysers as GATE Processing Resources

The GATE architecture describes Processing Resources (PRs) as implementations of algorithms. As such, the only restriction placed on a PR is that it can be executed. While this is a very flexible approach, it can benefit from refinements when used in the context of IE systems.

Processing Resources employed by IE systems will always have documents as input and will more often than not be used to process entire corpora rather than individual documents. This is common behaviour exposed by all PRs used for IE tasks, regardless of their actual function (e.g. part-of-speech tagging or named entity recognition). In order to support this in an uniform manner, we defined a specialised type of PR, called a Language Analyser, which extends the functionality of the standard PR by adding methods to access the document and corpus currently being processed. This was defined as a Java interface named gate.LanguageAnalyser that extends the gate.ProcessingResource interface provided by GATE. In order to facilitate implementations of Language Analysers, we also provide an abstract class called gate.creole.AbstractLanguageAnalyser that contains the common functionality for accessing the document and corpus fields. This class will be used as the base class for all PRs described in this thesis.

A diagram representing the classes mentioned above is shown in Figure A.2.

The main advantage of using Language Analysers as opposed to simple PRs is that they allow the uniform treatment of a set of documents grouped together as a corpus.

A.2.2 Execution Controllers

Information Extraction systems are usually implemented as pipelines made up of several modules: the input document passes in sequence through all of them, getting enriched with new information at each step. GATE provides support for implementing modules in the form of processing resources. In order to support the development of more complex applications, we needed
to add work-flow control functionality. This has been implemented in the form of execution controllers.

An execution controller provides functionality for grouping together a set of processing resources into compound applications and, similar to PRs, controllers also need to expose an `execute` method that is used to run all the contained processing resources. The functionality required for controllers is illustrated by the simplified representation of the `Controller` interface presented at the top of Figure A.3.

The simplest controller is one that implements a pipeline where several PRs are chained together and executed one after the other in the specified sequence. An implementation of this is provided under the name of `Serial Controller`, which holds a list of member PRs and has an `execute` method that simply calls the `execute` methods of the contained PRs. The specialisation from `Collection` to `List` for the data structure that holds the member PRs allows the `Serial Controller` to specify the right execution sequence.

Apart from being an architecture and framework, GATE aims to also provide an easy-to-use development environment. This means that most components
implemented for GATE can benefit from some graphical tool that allows them to be configured and customised. This is especially true for controllers, as they model applications that need to be modified frequently during the development phase. An interface we developed for **Serial Controllers** is presented in Figure A.4. It comprises three main areas:
• top left: a list of all Processing Resources that are currently loaded in the system and that can be used as members of the application.

• top right: a list of all the Processing Resources that have been added to the application. The two double arrows placed between these two lists permit the addition and removal of PRs from the application. The ‘up’ and ‘down’ arrows to the right can be used to re-order the member PRs.

• lower area: a table that allows the user to set values for the parameters that the Processing Resources take. This gets automatically populated whenever a new PR is selected in the list above. Suitable default values based on the meta-data associated with each PR type are used when actual values have not been provided.

Figure A.4: Configuration GUI for the Serial Controller

Serial controllers provide a method of executing a set of PRs in sequence over a document. However, in many cases, it is necessary to run processes over the documents contained in an entire corpus. Language Analysers are corpus-aware PRs and can be used for this. In order to take advantage of their capabilities we implemented the Serial Analyser Controller that extends the functionality of the Serial Controller (see Figure A.3 for a depiction of the class hierarchy). Given that GATE corpora are lists of
documents and any pipeline-style controller holds a list of PRs, the task of the Serial Analyser Controller is to run all the member PRs (or, more specifically, Language Analysers) over all the documents in a corpus. There are two strategies that can be applied:

- either run the whole set of PRs over each of the documents, moving on to the next document as soon as the full processing of the current one is finished, or,

- run each PR over all of the documents and start again with the next PR when the documents in the corpus have been exhausted.

In terms of the final result achieved these two strategies are equivalent but, as will be shown next, there are practical reasons which made us prefer the former.

GATE corpora are not limited in size, they can contain an indefinite number of documents. In practice, this number is limited by the number of documents that can be loaded in the computer's memory at the same time, which is not very large (e.g. a few hundred typical web pages). To circumvent this restriction, GATE provides the concept of DataStore—a mechanism for storing in a persistent manner any type of Language Resources, including documents and corpora. This allows users to create corpora that are as large as they can fit onto their computer’s hard drive which, while still limited, usually exceeds normal requirements. For corpora stored in a DataStore, GATE provides a persistent corpus implementation that can transparently load documents from the DataStore when they are needed and has API calls for unloading documents from memory when not required any more.

During the design of the Serial Analyser Controller we needed to cater for the use of serial corpora and datastores. The operation of loading documents from the datastore is expensive in terms of time so it needs to be used sparingly to preserve efficiency. When comparing the two execution strategies described above, it becomes apparent that in the first case each document will need to be loaded and unloaded only once during the whole process, while the second strategy would require each document to be loaded and unloaded once for each processing resource contained in the controller.
Infrastructural Support for Information Extraction

Following these considerations, the logic implemented by the analyser controller is described in the following steps:

1  for each document
2      if document not loaded, load document;
3      for each PR in the controller
4          process current document with current PR;
5      if current document was loaded at line 2, then
6          save the changes to the document into the datastore and
7          unload the document;
8      go to the next document.

In terms of user interface, Serial Analyser Controllers benefit from a configuration tool that is very similar to the one used for simple Serial Controllers. The only differences are: 1) there is a new drop-down box that allows the user to select a corpus to be used for processing, from all the corpora loaded in the system; 2) only PRs that are Language Analysers are allowed as members of the controller; and 3) for all the member PRs, the parameters corpus and document are not available to be modified by the user, as they are set automatically by the controller.

Serial Analyser Controllers are capable of processing all the documents from a corpus in a uniform manner. This can improve productivity during the development phase of an information extraction system, but can also cause problems when the documents in an input corpus are not homogeneous in nature. For instance, let us consider a corpus obtained from crawling a set of web sites: in this case the large majority of the documents will be web pages written in some version of the HTML language. However, other types of documents might be included occasionally, for example PDF pages are quite frequently found on the Internet. If we also assume that HTML and PDF documents need to be treated differently (e.g. a different type of analyser needs to be run to extract data regarding the page formatting) then the system developer is faced with a problem. A similar problem occurs when the input corpus contains documents in different languages, for instance. To address these types of situations, we have added a new type of controller that is able to dynamically change the execution flow according to the features
of the current document. We call this a **Conditional Controller**, and we provide implementations that provide this functionality for both **Serial Controllers** and **Serial Analyser Controllers**, see Figure A.3.

A **Conditional Controller** associates with each of the member PRs a **running strategy** that is used to decide whether a particular PR should be run or not. In the case of the **Conditional Serial Controller**, the decision is manually set by the user before executing the controller, and can be changed between runs using the graphical interface. This can be useful during the development phase for temporarily disabling the execution of some components in the IE system in order to perform experiments. However, the real benefit is obtained in the case of **Conditional Serial Analyser Controllers**, where the decision is based on the value for a particular feature on the current document, and is dynamically calculated at execution time. For example, one can envisage a system where the input comprises documents in different languages. The first analyser in the pipeline is used to identify the document language which is then marked by means of a feature on the document. All the subsequent PRs can be run or not depending on whether they are capable of processing documents in that particular language.

This functionality is reflected in the graphical user interface depicted in Figure A.5, which shows the user interface associated with a **Conditional Serial Analyser Controller**. This allows us to illustrate the changes to the interface required for implementing the analyser functionality as well as those necessary for the conditional behaviour. Comparing Figure A.5 with the interface for the **Serial Controller** shown in Figure A.4, there are a few notable differences:

- the bullet points associated with each PR now use a colour code to reflect the corresponding running strategy: green for PRs that are always run, red for PRs that are excluded from execution and yellow for PRs that are run conditionally;

- when a PR is selected, there is an additional panel that allows the editing of the execution strategy. This lets the user select the execution mode for that PR.
the corpus selection drop-down box is visible, which indicates that this is an analyser controller;

- the document parameter for the selected PR is not shown any more, as the analyser controller will set it automatically.

This hierarchy of controller types as well as their associated user interfaces enable rapid development for IE systems. They allow the definition of various pipeline architectures, the processing of large corpora through the use of datastores, and the flexibility of dynamically modifying the system configuration depending on the input by using conditional controllers. All these facilities are aimed at helping the system engineer achieve the desired results as quickly as possible.

This section described the various types of execution controllers implemented for use with GATE. From here on we shall be using the term of *application*, or simply *controller*, to refer to an IE system where the actual type of controller used is not specified or is irrelevant.
A.2.3 Application Persistency

All the work in defining and fine tuning an IE application is performed using the various graphical tools presented to the user. Once the system has reached a satisfactory state it needs to be packaged in a suitable manner so that it can be deployed for its intended application or preserved for future refinements. This requires storing the system architecture both in terms of the modules that are being used as well as their individual configurations.

Processing Resources are configured through two sets of parameters: init-time parameters that are used when the PR is first created and run-time parameters that make up the options used during execution. Both of these sets contain parameters that consist of a name, a type and a value; the name is a string value, the type is an arbitrary Java class and the value is an object of the class used as type. To store the configuration of a whole IE system one needs to preserve the following bits of information:

- The list of plugins currently loaded in GATE. Plugins are sets of language processing components that are not included in the core framework but are defined by an external source. These need to be preserved as the components from external plugins are likely to be used by the application being stored.

- The type of controller used by the system as well as any parameters used by the controller (e.g. the list of running strategies in the case of conditional controllers).

- The list of Processing Resources that are part of the application.

- All the init-time and run-time parameters for the PRs.

GATE resources accept all types of Java objects as parameters. In order to allow the saving of parameter values to files (for application persistency) we need to restrict the permitted values to either atomic types or instances of serialisable classes, i.e. classes that provide a mechanism for converting an arbitrary instance into a binary value. This restriction is not very hard as most standard Java classes are already serialisable and there is a default mechanism for serialising arbitrary user-created classes. Custom classes
Infrastructural Support for Information Extraction

that require a special mechanism for serialisation can easily implement it if required.

Using the Java serialisation mechanism makes it easy to store the creation metadata and configuration for any GATE resource by simply saving the set of associated init-time parameters, run-time parameters and features. The only exceptions are parameters and features that take either URLs or other GATE resources as values, which require special treatment as explained below.

URLs are used extensively in GATE for parameters pointing to files on disk, e.g. the set of files containing the phrases to be recognised by a gazetteer. While they could be serialised using the standard mechanism, this would create problems when moving saved applications from one computer to another as the actual locations of the resource files are unlikely to be identical on different machines. The solution we chose to this problem is to store all local URLs (i.e. URLs with the file: protocol) as paths relative to the file where the application is being saved. This way, saved application become portable between machines, the only condition being that the application file and the associated resource files remain in the same relative position, which is easy to achieve if they are kept within the same top directory for instance. Preparing a GATE application for persistency will now consist of creating a new top directory, moving all the required resource files inside that directory, making sure that all parameters and features point to files under the top directory and saving the application itself in a file inside the same top directory. The application directory can then be moved to any other computer running GATE where the saved application can be opened and used.

Another special case occurs when one of the parameters or feature values is a GATE resource itself. In such situations the default serialisation mechanism is overridden to only save the metadata required for recreating the resource rather than serialising the actual Java object represented by the resource. If the GATE object being saved as a parameter value has already been saved as part of the application, then a simple reference to the existing representation is used instead. This preserves identity by avoiding the creation of duplicate copies of the same resource, and it also saves storage space and time during
serialisation and deserialisation.

A.3 Tools for Manual Annotation

Manually annotated data is important for the development of Information Extraction systems. In the case of the knowledge engineering approach, it is used to drive the development process by taking a data-driven approach that directs effort first toward the most frequently occurring phenomena. When choosing the learning approach, the manually annotated text is the main resource used during development and the quantity and quality of available data directly influences the highest performance that the finished system can possibly reach. For both the knowledge engineering and learning approaches, manually annotated data (the gold standard) is used for evaluating the current performance of the developing system and for directing future efforts toward the problem areas identified by an analysis of the errors that are systematically made.

![Figure A.6: The document editor.](image)

While manually annotated text is useful, it is also fairly expensive to pro-
duce, as it requires human effort, by its very nature. Manual annotation is a very repetitive task, which makes it tedious and error-prone. To mitigate these issues, we built a set of graphical tools that aim to make the annotation process easier and more efficient. The main user interface for viewing and editing textual documents is the document editor presented in Figure A.6. It consists of up to three panes that display different views of the document data:

- the “textual view” shown in the central pane displays the text with highlights in different colours for different types of annotations;

- the “annotation sets view” included in the vertical side pane shows in a tree-like structure all the annotation sets present on the document and the annotation types existing in each annotation set. The first annotation set is always the default one, which has no name, followed by an arbitrary number of named sets. The user can select any annotation type belonging to any set causing the corresponding textual spans covered by the selected annotations to be highlighted in the textual view;

- finally, the “annotations list view” shown in the horizontal pane displays a list of all the currently selected annotations, including details such as the start and end character offsets in the text and the annotation features.

Either of the component panes can be shown or hidden by selecting the buttons at the top of the document editor.

The implementation of the document editor is based on a plug-in architecture; all content panes are GATE Visual Resources (VRs) defined as CREOLE plugins. The three panes described above are part of the actual document editor, but additional views can be implemented and provided as plugins. During initialisation, the editor will enumerate all types of VRs known to the system, and will automatically instantiate the ones that are compatible with it. In the example shown in Figure A.6 an extra button is present for a co-reference editor that was automatically discovered and initialised due to its CREOLE configuration.
Beside the facilities for viewing the document data, the main function of the document editor is that of manual annotation of textual data, provided through the integrated annotation editor as illustrated in Figure A.7. The annotation editor is implemented as a pop-up window that is displayed whenever the mouse hovers over an existing annotation or over a selected piece of text. The main components of the annotation editor are:

- at the top, a bar with buttons for modifying the left boundary of the annotation, for deleting the annotation and for moving the right boundary. The arrow buttons for moving the annotation boundary can change the offset by one character when clicked or by five or ten characters if modifier keys are also pressed.

- under the bar of buttons, there is a drop-down box for setting the annotation type. The list is pre-populated with values representing all the annotation types present in the current annotation set. If neither of those values is appropriate, the user can type in a new annotation type.
finally, a fully editable tabular view is presented for the annotation features. The user can directly change the name or value for any feature, or they can completely delete a feature or add a new one.

The behaviour of the annotation editor was designed in a way that minimises the number of mouse clicks and the size of the mouse movements required. Creating a new annotation without features, which is the most frequent operation, requires only selecting the right piece of text, holding the mouse pointer over the selected span (which causes the annotation editor window to show) and selecting the right annotation type from the drop-down box. Occasional errors in selecting the right span can easily be corrected through the use of the boundary move buttons. As a further optimisation, the editor remembers the last annotation type used and uses it as the default value for newly created annotations. This allows the user to quickly create all annotations of the same type in sequence by simply selecting the span and holding the mouse over it, without needing to perform any clicks inside the editor window, as the right annotation type would have already been selected. The editor window disappears automatically if no mouse activity is detected inside it after a short period of time, and the user can then move on to the next annotation of the given type. These automations speed up the manual annotation process. Encouraging the annotator to create all annotations of one type in sequence can also have a positive influence over the correctness of the result, as it reduces the amount of context switching in the mind of the annotator: it is easier, for example, to mark all “Person” annotations first and then move on to “Location”s rather than annotating both in one go.

GATE uses XML schemas, called Annotation Schemas for defining the set of features and values that are permitted or required for an annotation type. Annotation schemas are implemented as language resources, and an arbitrary number of them can be loaded into the system at any one time. The annotation editor is aware of the schemas that are known, and uses them to populate the features editing component: optional and required features are marked as such and the set of permitted values are presented to the user for selection. When no appropriate schema for a given annotation type is found, the editor works in unrestricted mode, allowing the user to
enter arbitrary values.

We also produced an alternative annotation editor component which con-
strains the available annotation types and features much more tightly, based
on the annotation schemas that are currently loaded. This is intended to be
used for annotation exercises where the annotator is forced to adhere to the
schema specifications. Because the users are more restricted in the options
available to them, this allows for a more streamlined interface, with the side
effect of speeding up the annotation process.

![Restricted Annotation Editor](image)

**Figure A.8: The restricted annotation editor.**

The modified annotation window is shown in Figure A.8. As can be seen, the
drop-down boxes in the standard editor have now been replaced with groups
of buttons, one group each for selecting the annotation type, and the value
for each of the features. This makes the whole interface easier to use and
it reduces the scope for errors, which is particularly useful when annotating
large quantities of data or for use by less-skilled users.

Annotated documents are usually stored in corpora, which are represented as
lists of documents. A simple corpus editor was implemented and is shown in
Figure A.9. The buttons at the top can be used to add or remove documents
and to modify the order of the documents in the corpus.

In most cases, textual data is stored as files organised in directories on disk.
Infrastructural Support for Information Extraction

Figure A.9: The corpus editor.

Figure A.10: Populating a corpus.
It is useful to be able to create a GATE corpus starting from a set of files on disk. Normally this would require creating a new corpus, loading all documents one by one, and then adding them to the corpus. This can be a lengthy process, especially when a large number of files are involved, so, in order to address this issue, we implemented a graphical tool for populating a corpus from a given set of files. The interface, presented in Figure A.10, allows the user to select a root directory and fill an existing corpus with documents created from files found in that directory. The files to be loaded can be filtered by extension and the character encoding to be used can be specified. Recursive traversal of all sub-directories is also possible.

A.4 Support for Evaluation

Quantitative performance evaluation is an integral part of the development process for information extraction systems. Given a task specification, the first step is to create a manually annotated corpus that reflects the type of results expected from the finished system. This corpus is then used for creating the first prototype of the system, either by rule authoring in the case of knowledge engineering or training of models in the case of the learning approach. Sometimes, especially in the case of competitions, a part of the manually annotated corpus is kept and is used at the end of the process for testing on unseen data. The development cycle then continues by testing the current prototype to evaluate its performance, analysing the errors made, and refining the rules or feature sets in order to address the most frequent mistakes. The development process stops when the system reaches a particular level of performance that was set up from the outset or when no further improvement is foreseen given the resources available. Depending on their application, some IE systems are continually refined throughout their life in order to keep up with changes in the input or requirements. Quantitative performance evaluation is essential for both assessing the improvements obtained after each refinement step and for establishing when the current performance level is adequate and the development can stop.

As explained in Section 2.4, the most frequently used evaluation metrics are those of Precision and Recall, which are sometimes combined in a single
Infrastructural Support for Information Extraction

To assist the development of IE systems, a facility for calculating these values was required. Also needed was a method for efficiently browsing the differences between the current results and the gold standard. To address these two needs we developed a tool, called *Annotation Diff*, that shows the differences between two sets of annotations in manner similar to various graphical differencing tools employed in software development. The user interface of the tool is pictured in Figure A.11.

The *Annotation Diff* tool compares two sets of annotations, one called *Key* representing the gold standard, and the other called *Response* which is the one being evaluated. These two sets are obtained from existing annotation sets on documents loaded in the system; typically the two versions of annotation are stored on different annotation sets for the same document, though this is not a requirement. The upper part of the user interface is used to select which annotation sets from which documents are to be used for performing the calculations. Comparing annotations only makes sense for annotations of the same type, so the user is also asked to select the annotation type. Two annotations, one key and one response, are considered equal if they cover the same span, have the same type and the same features. Some annotations, especially the automatically generated ones, have

Figure A.11: The *Annotation Diff* tool.
extra features created for storing temporary or debug information. Because of this, only the set of features present on the key annotations are used for comparison, any extra features on the response annotations being ignored. For additional flexibility, the user can choose to completely ignore all features when comparing annotations, or only use a specified sub-set. Finally, the user can set the value of the relative weight for Precision and Recall used when calculating the F-Measure value. After all the parameters are set, the user can request the tool to perform the calculations. When this is done, the results are shown in the central table. Each table row presents a pairing between a key and a response annotation. Annotations that are considered equal are shown side by side; annotations that are only partially overlapping are considered partially correct and are also shown side by side, but on a different background colour; unpaired key and response annotations are shown independently and on different backgrounds. Below the main table, the values for the calculated metrics are displayed. This includes the numbers of correct, partially correct, missing and false positive annotations (missing annotations are the ones present in the key set but with no corresponding response; false positives are unpaired responses), the values for strict, lenient and average Precision, Recall and F-Measure. Lenient values are calculated taking partially correct matches as correct, while average values are averages of strict and lenient measures. The results table can be sorted based on the values of any column, and columns can be reordered or hidden. The current configuration of the results table, together with the numeric values, can be exported as an HTML file for persistent storage.

The algorithm used for computing the differences starts by considering all possible pairings between all key annotations and all response annotations. All these potential pairings are then sorted according to a scoring where correct matches are better than partially correct ones, which are better than missing and spurious matches. The actual scoring used is slightly more complex than this, as it takes into account the fact that once key and response annotations are paired, they cannot take part in other pairings, so the score reflects both the type of the current pairing and the set of other potential pairings that are being blocked by it. The pairings are then consumed in a greedy fashion, starting with the highest scoring ones and taking care that once a pairing is used, all other remaining pairings that are
mutually exclusive with it are removed, as any key or response annotation can only be used once for calculating the score. This way, the algorithm ensures that the best possible score is given to any response annotation. If, for example, several response and key annotations can be seen as either correct or partially correct matches, then the set of key-response pairings that the algorithm makes is the one that maximises the score.

A.5 Infrastructure for Multilinguality

It is often thought that the character sets problem has been solved by the arrival of the Unicode standard. This standard is an important advance but, in practice, the ability to process text in a large number of the world’s languages is still limited by:

- incomplete support for Unicode in operating systems and applications software;
- languages missing from the standard;
- difficulties in converting non-Unicode character encodings to Unicode.

This section describes the infrastructural problems associated with processing multiple languages, and describes the GATE Unicode Kit which we implemented to address some of these problems.

A.5.1 Unicode-compliant Language Resources

Documents in GATE are typically created starting from an external resource such as a file situated either on a local disk or at an arbitrary location on the Internet. Text needs to be converted to and from binary data, using an encoding (or charset), in order to be saved into or read from a file. There are many different encodings used worldwide, some of them designed for a particular language, others covering the entire range of characters defined by Unicode.

All operating systems have a default character encoding that is used whenever textual data is read from or written to files. While this is appropriate
Figure A.12: Parameters for loading a document.

Figure A.13: A Chinese document processed in GATE.
for most general purpose applications, in the case of language engineering applications, such as GATE, one cannot assume that all input files will be encoded using the platform default. Because of this, GATE was adapted to take advantage of the facilities provided by the Java platform giving it access to over 100 different encodings, including the most popular local ones, such as ASCII and ISO 8859-1 in Western countries or ISO-8859-9 in Eastern Europe, and some Unicode ones e.g. UTF-8 or UTF-16. We made sure that in all cases where a textual file needs to be accessed there is a parameter available that allows the user to set the character encoding to be used.

Figure A.12 shows the parameter set used when creating a new document from a file. As can be seen, the second parameter (in alphabetical order) is the one for the encoding. The default value for the encoding is blank and, when left unmodified, the platform default will be used, thus mimicking the default behaviour of other applications.

### A.5.2 Processing Resources and Unicode

The use of the Java platform implies that all processing resources that access textual data will internally use Unicode to represent it, which means that all PRs can be used for text in any Unicode-supported language. Most PRs, however, need some kind of linguistic data in order to perform their tasks (e.g. a parser will need a grammar) which in most cases is language specific. In order to make the algorithms provided with GATE (in the form of PRs) as language-independent as possible, and as a good design principle, there is always a clear distinction between the algorithms (presented in the form of machine executable code) and the corresponding linguistic resources (which are typically externalised as files). All PRs have access to the available charset decoders when loading external resources so these resources can use any supported encoding. This allows a gazetteer list for instance to contain localised names.

### A.5.3 A Unicode-Aware Graphical Interface

The third type of resource provided with GATE is the Visual resource used in constructing user interfaces, for the visualisation and editing of data, as
well as for control of the execution flow. A Unicode-enabled graphical user interface (GUI) needs to address two main issues: the capability to display text and the ability to accept text entered in languages other than the default one.

For displaying Unicode text, GATE makes use of the standard Java text rendering engine. This means that a wealth of languages are already catered for, while new additions brought in by new developments to the Java platform will automatically be available as they are released. Figure A.13 shows a Chinese document that has been opened and annotated in the standard GATE interface.

While support for displaying Unicode text is provided to a large extent by the underlying platform, the same is not true as regards the input of Unicode text. Many platforms provide some support for localisation but that is not always very comprehensive and is often not Unicode compliant, which makes it difficult for Java applications to make use of it. Also, the level of support provided varies widely between platforms. This problem is addressed by the GATE Unicode Kit (GUK), a dedicated and extensible library for text input.

![Figure A.13: A Chinese document in the standard GATE interface.](image)

GUK plugs into the underlying Java virtual machine and provides a variety of input methods dedicated to different languages.\(^2\) These input methods are

\(^2\)Currently 17 languages are supported through 30 input methods.
used to map the input from the system keyboard into what the input should have been had a localised keyboard been installed. The mapping is defined in a simple text file which allows users to add new languages or to adapt current input methods to their particular needs. Complex many-to-many mappings are supported, e.g. when several key presses are needed to emit one output character, or conversely, when one key press leads to several characters being produced. In order to facilitate the input of text on a keyboard layout that the user may not be familiar with, a virtual keyboard map can be displayed whenever an input method is activated. When multiple key presses are used to enter single characters, as is the case with the Korean keyboard displayed in Figure A.14, then the keys that can be pressed next (given the previous presses) are highlighted.

Apart from the input methods, GUK also provides a simple Unicode-aware text editor, which is useful in a cross-platform environment as not all operating systems offer one by default. Besides text visualisation and editing facilities, the GUK editor also performs encoding conversion operations. Figure A.15 illustrates the GUK Unicode editor showcasing some of the supported scripts.

![Gate Unicode Editor - Unicode Sampler.txt](image)

Figure A.15: Some of the Unicode-supported scripts.
A.6 Processing Resources for Information Extraction

Many parts of an IE system are typical NLP components for which there are already known algorithms that have been tried and are known to be robust and perform within given accuracy levels. Much of the work in designing a new IE system consists of prototyping and trying various configurations with different tool sets and parameter values. During this phase, it is useful to have access to a set of ready-made implementations of such algorithms that can be used as building blocks. This section describes the set of basic NLP tools that were implemented in order to support the system developer and are made available in the form of the ANNIE plugin for GATE.

These components can be used as-they-are, as parts of new IE systems, or can provide a useful starting point for other implementations that are customised to deal with particular types of data. A principle adhered to during the development of these tools was the separation of data and logic: all modules comprise Java code (which implements the algorithm) and linguistic data that describe the rules used during execution. This allows the behaviour of the processors to be altered by modifying the rule sets without needing modifications to the source code which would require recompilation and lead to various other complications. This means that the tuning of the system can be performed by computational linguists that are not necessarily Java programmers.

Taken together, all the tools in the ANNIE plugin make up a baseline IE system that performs Named Entity Recognition. The overall architecture of ANNIE is described in Figure A.16.

ANNIE uses a Serial Analyser Controller to define a pipeline containing the following Processing Resources:

- a Unicode Tokeniser used to split the input text into tokens and space tokens.
- a Gazetteer Lookup component that identifies key words and phrases.
• a **Sentence Splitter** that breaks the input into sentences.

• a **Part of Speech Tagger** that associates a part of speech with each input token.

• a **Named Entity Recogniser** that finds entities in the input based on patterns relying on the annotations created by the previous modules.

Each of these modules are described in details in the following sections.

### A.6.1 The Unicode Tokeniser

The first Processing Resource to interact with the input text in ANNIE is the tokeniser. This is built around the Unicode ([Unicode Consortium 96](#)) standard in order to provide generality and extensibility to new languages. The role of the tokeniser is to split the text into simple tokens and to provide some basic information about their types. It classifies tokens into numbers, punctuation, symbols or words and, in the case of words, provides some information about their orthography (e.g. with an initial capital, all upper
The design of the tokeniser is generic enough that it can be applied unmodified to input texts in a variety of languages.

Running the tokeniser over a document will generate a group of annotations of type **Token** or **SpaceToken** which will never overlap and will cover the entire content of the document (see Table A.1 for an example).

<table>
<thead>
<tr>
<th>Text</th>
<th>Annotations</th>
</tr>
</thead>
<tbody>
<tr>
<td>On March the 3rd 2002...</td>
<td><img src="#" alt="Table A.1: A tokenisation example" /></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type</th>
<th>Start</th>
<th>End</th>
<th>Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>Token</td>
<td>0</td>
<td>2</td>
<td>kind=word, orth=upperInitial, string=&quot;On&quot;</td>
</tr>
<tr>
<td>SpaceToken</td>
<td>2</td>
<td>3</td>
<td>kind=space, length=1 string=&quot; &quot;</td>
</tr>
<tr>
<td>Token</td>
<td>3</td>
<td>8</td>
<td>kind=word, orth=upperInitial, string=&quot;March&quot;</td>
</tr>
<tr>
<td>SpaceToken</td>
<td>8</td>
<td>9</td>
<td>kind=space, length=1 string=&quot; &quot;</td>
</tr>
<tr>
<td>Token</td>
<td>9</td>
<td>12</td>
<td>kind=word, orth=lowercase, string=&quot;the&quot;</td>
</tr>
<tr>
<td>SpaceToken</td>
<td>12</td>
<td>13</td>
<td>kind=space, length=1 string=&quot; &quot;</td>
</tr>
<tr>
<td>Token</td>
<td>13</td>
<td>14</td>
<td>kind=number, string=&quot;3&quot;</td>
</tr>
<tr>
<td>Token</td>
<td>14</td>
<td>16</td>
<td>kind=word, orth=lowercase, string=&quot;rd&quot;</td>
</tr>
<tr>
<td>SpaceToken</td>
<td>16</td>
<td>17</td>
<td>kind=space, length=1 string=&quot; &quot;</td>
</tr>
<tr>
<td>Token</td>
<td>17</td>
<td>21</td>
<td>kind=number, string=&quot;2002&quot;</td>
</tr>
<tr>
<td>Token</td>
<td>21</td>
<td>22</td>
<td>kind=punctuation, string=&quot;.&quot;</td>
</tr>
<tr>
<td>Token</td>
<td>22</td>
<td>23</td>
<td>kind=punctuation, string=&quot;.&quot;</td>
</tr>
<tr>
<td>Token</td>
<td>23</td>
<td>24</td>
<td>kind=punctuation, string=&quot;.&quot;</td>
</tr>
</tbody>
</table>

The Unicode Tokeniser is built as a finite state machine (FSM) that uses the character categories defined in the Unicode standard as its input symbols. The FSM machinery was chosen as the underlying implementation because it is particularly suited to text processing tasks: it provides a flexible way
of defining the tokenisation rules, has good algorithms for optimising the execution speed and uses relatively small amounts of memory. The use of the Unicode categories as input alphabet allows for a relatively simple set of rules to cover adequately a large number of languages.

The underlying FSM of a tokeniser is defined by a set of rules, each of them being composed from a regular expression on the left hand side and an annotation template on the right hand side. The regular expression defines a pattern of characters, named by their respective Unicode categories, while the annotation template is used to generate new annotations when the pattern described on the left hand side matches the input. See Figure A.17 for an example of a tokeniser rule which detects words containing only letters and dashes and that start with a capital letter. Each such rule is converted into an FSM designed to recognise the regular expression pattern, and then the full FSM for the tokeniser is constructed as a disjunction of all the FSMs defined by the rules. The default rule set contains 23 rules that recognise many types of words, whitespace patterns, numbers, symbols and punctuation, and which can handle many world languages without any modifications.

```
UPPERCASE_LETTER
(LOWERCASE_LETTER|DASH_PUNCTUATION)*
> Token;orth=upperInitial;kind=word;
```

Figure A.17: A tokeniser rule.

For situations not handled by the current tokeniser, new rules can easily be added, extending it to new languages.

The modular architecture of GATE allows for the customisation of the tokeniser results when a particular language requires additional processing, by simply adding another processing resource after the tokeniser in the execution chain. For instance, in the case of English text, a simple JAPE transducer (see Section 3.2) is used which adapts the output of the tokeniser for the needs of the part-of-speech tagger.
A.6.2 The Gazetteer Look-up Component

Many IE tasks rely on a large number of key words or key phrases being identified in text. The lists of relevant terms and phrases are usually called gazetteers, and may include names of known entities, such as person names, geographical locations or organisations, or several types of indicators such as the days of the week, names of the months, personal titles, currency names, etc. Recognising all occurrences of these phrases in text is a simple pattern-matching problem. However, the fact that the number of patterns sought is usually very large, ranging from tens of thousands to millions, requires some special implementation that is efficient both in terms of speed and memory usage.

Because of these requirements, we chose to implement a component for gazetteer lookup using finite state machinery. One reason for this choice was that a finite state machine can be used to very efficiently search for a large number of patterns in parallel. As important was the fact that it can compress the memory representation of the input patterns by sharing the same memory for common prefixes. This reduces the amount of memory used without sacrificing any execution speed. Unlike the case of the tokeniser, here the input symbols are the actual characters.

A gazetteer lookup component is created starting from a set of lists as plain text files, each of them representing a class of names, such as names of cities, organisations, days of the week, etc. If necessary, the lists can be encoded using Unicode, which allows for localised names of people or locations for instance. Each list has up to three features associated with it: a majorType, a minorType and a language. For example a list of city names might have a majorType of ‘Location’ and a minorType of ‘city’.

The gazetteer finite state machine is built at initialisation time starting from the list of phrases that need to be recognised. Because the lists get compiled into a minimised finite state machine, the gazetteer can easily handle very large sets of entries. Using the default entry list provided with GATE, which contains around 80,000 entries, the gazetteer lookup component is initialised in under one second and can process input text at a rate of around one million characters per second, running on a typical laptop.
At run time, all occurrences of the terms in the gazetteer lists are marked by annotations of type Lookup having the features associated with the list where the term was included. Runtime options are provided that can cause the gazetteer to ignore capitalisation or to recognise only whole words.

### A.6.3 The Sentence Splitter

The sentence splitter is a cascade of finite-state transducers which segments the text into sentences. The splitter uses a list of known abbreviations to help distinguish sentence-marking full stops from other kinds. Each sentence is marked by an annotation with the type Sentence, while each sentence break (such as a full stop) is given a Split annotation. Splits can either be internal, such as a punctuation sign which is part of the sentence, or external in the case of carriage return characters which are also used to break the input text. Because the splitter is intended to work on all types of input, it does not make any assumption with regard to the grammatical correctness of the text and makes use of multiple indicators for identifying sentence splits, such as empty lines, punctuation even when not followed by a capital, repeated punctuation marks like “?!?!”.

The main reason for including a sentence splitter was for the benefit of the POS tagger, which was trained on sentences and thus can be expected to perform better if its input is also separated into sentences.

The Split annotations are also a useful side-product of the sentence splitting process. When performing IE, they can be used to block the matching of longer text segments that span over them; for example it is unlikely that a person name would ever start in one sentence and end in the next, thus spanning a sentence split. The Split annotations can be used to stop such spurious matches.

### A.6.4 The Part of Speech Tagger

A Part of Speech Tagger classifies input words according to their morphological category. This is a classical problem in NLP and there are a number of algorithms that can be used for addressing it. All state-of-the-art implementations have very high levels of accuracy, comparable to that of
a human annotator. When the input text differs greatly from the types of
texts used during the training of the POS tagger, the quality of the results
usually suffers but the performance levels are still quite high, usually above
80%.

POS tagging is very useful for Information Extraction because it provides
an abstraction level above the input words. A single pattern based on POS
values can cover a large number of actual word combinations.

The actual tagger chosen for inclusion with the ANNI tools is Mark
Hepple’s tagger [Hepple 00] which is a modified version of the Brill tagger
[Brill 95]. This was chosen because it is a state of the art tagger and because
the author gave us permission to freely distribute the resulting resource to
the research community.

The tagger produces a part-of-speech tag as a feature on the annotations
produced by the tokeniser for each word or symbol. The list of possible
tags is the one used in the morphological annotation of the Penn Treebank
[Marcus et al. 94], a corpus widely used by the computational linguistics
community.

The tagger uses a default lexicon and ruleset (the result of training on a large
corpus taken from the Wall Street Journal). Both of these can be modified
manually if necessary. Two additional lexicons exist, one for texts in all
uppercase and one for texts in all lowercase, which are useful for example in
the case of text obtained from automatic speech recognition systems, which
provide no capitalisation.

The advantage of using a variant of the Brill tagger is that it is a widely
used algorithm and rulesets already exist for a wide variety of languages.

A.6.5 The Named Entity Recogniser

The Named Entity Recogniser consists of a set of transducers that run in
cascade one after another. Each transducer recognises particular annotation
patterns and generates new annotations as a result. All annotations
produced by the upstream modules, such as the Tokeniser, POS Tagger or
Gazetteer lookup, are available as input and the annotation patterns used
can combine them freely. The underlying technology used is JAPE, and is
described in detail in Section 3.2.

The ANNIE entity recogniser can be used as-is, or it can provide a useful
starting point for customisations to particular application needs.

A.7 Working with Ontologies

An increasing number of NLP projects make use of taxonomic data struc-
tures and ontologies. The use of NLP techniques for (semi-)automatically
generating Semantic Web meta-data is also a growing trend. The advance-
ments in the Semantic Web research area have led to a variety of standards
for representing ontologies, and an increasing number of tools and program-
ing libraries for managing ontologies are becoming available. All this un-
derlines the need for NLP systems to access ontological information and has
led to the addition of support for ontologies in GATE.

The various ontology representation formalisms (such as RDF-Schema
[Lassila & Swick 99], OWL and its variants [Dean et al. 04], or DAML-OIL
[Horrocks & van Harmelen 01]) have their advantages and disadvantages
as well as their idiosyncrasies. Rather than attempting to choose one of
the formalisms based on what can only be subjective criteria and running
the risk of obsolescence when that particular formalism falls out of grace
with the research community or is superceded by a newer one, the GATE
ontology support is aiming at providing an abstraction layer between the
actual representation mechanism and the NLP modules making use of it.
It consists of an in-memory data model for ontologies, an API providing
access to that representation, a visual resource displaying the information,
and input/output capabilities for accessing files containing ontologies using
various standards. This approach has well-proven benefits, because it
enables each application to use this format-independent model when dealing
with ontologies, thus making the application immune to changes in the
underling ontology formats. If a new format needs to be supported,
the application can automatically start using ontologies in this format, by
simply including the correct tool that converts the format into the common
model. From a language engineer’s perspective, the advantage is that they
only need to learn one API and model, rather than having to deal with many different ontology formats. This approach is similar to the way we deal with document formats.

A.7.1 Data Model for Ontologies

In order to work as an abstraction layer, the GATE ontology implementation only supports those features that are common to all formalisms, which are also the features most widely used. All the information that is specific to a given representation model and cannot be represented in GATE is ignored. Currently, the ontology data model has support for hierarchies of classes, hierarchies of properties, and instances (also known as individuals).

Hierarchies of classes

The central role in the ontology data model is played by the class hierarchy, also known as a taxonomy. This consists of a set of classes linked by subClassOf and superClassOf relations. Classes have a name and a URI; in most cases the name is the local part of the URI, though this is not enforced. Class names within an ontology must be unique. Classes can also have comments which are used to explain their intended meaning.

All classes can have a set of superclasses and a set of subclasses which are used to build the class hierarchy. The subClassOf and superClassOf relations are transitive, and methods are provided by the API for calculating the transitive closure for each of these relations given a class. The transitive closure for the set of superclasses for a given class is a set containing all the superclasses of that class, as well as all the superclasses of its direct superclasses, and so on until no more superclasses are found. This calculation is finite, the upper bound being the set of all the classes in the ontology. A class that has no superclasses is called a top class. An ontology can have several top classes. Although the GATE ontology API can deal with cycles in the hierarchy graph, these can cause problems for processes using the API and probably denote an error in the definition of the ontology. Care should be taken to avoid such situations.

Instances

147
Instances are objects that belong to classes. Like classes, they have a **name** and a **URI** and can have **comments**. One instance can belong to one or more classes and can have properties with values. There are API methods provided for getting all the instances that belong to a given class or the property values for a given instance.

**Hierarchies of properties**

The last part of the data model is made up of hierarchies of properties that can be associated with objects in the ontology. GATE defines three types of properties:

**Object Properties** are properties that are associated with an ontology instance and have an instance as value.

**Datatype Properties** are properties that are associated with an ontology instance and can have any Java object as value.

**Generic Properties** are associated to any ontology object, be it a **class**, an **instance**, or another **property**, and have any Java object as value.

From an OWL/RDF perspective, object and datatype properties are similar to the property types as defined by OWL variants, while the generic properties are intended to model the properties defined by RDF.

Unlike some other representation models, in GATE, properties do not ‘belong’ to classes, they are instead first-class citizens of the data model. The specification of the type of objects that properties *apply* to is done by means of **domains**. A **domain** for a property is a set of ontology classes. The classes listed in the domain should be seen as a set of restrictions: for a property to be applicable to an instance, that instance needs to belong to all the classes in the property’s domain. A property with a domain that is an empty set can apply to instances of any type (i.e. there are no restrictions given).

Similarly, the types of values that a property can take is restricted through the definition of a **range**. Object properties take instances as values, so their ranges are also sets of classes; the manner in which the range restrictions are interpreted is similar to the case of the domain. Datatype and generic properties can have as value any Java object. Their ranges are sets of Java
Classes (objects of type `java.lang.Class`) which means they can only have values that implement those classes. In most cases, such a range will either be empty (allowing any type of object) or contain a single Class value (the most specific Class from the class hierarchy that is appropriate). Having several classes that are not in a hierarchical relation effectively blocks that property from accepting any value, while including several classes that are hierarchically linked (derived from one another) is superfluous, as only the most specific one is significant. Of course, a range can contain a set of objects representing interfaces, in which case non-singleton sets make sense.

Although the data-model and API permit the use of any kind of values for datatype and generic properties, if input/output to XML serialisations (such as XML-based OWL or RDF) is required, then the allowed values are limited to the set of types representable in those serialisations (usually the ones defined by the XML-Schema specification).

All properties can be marked as functional properties, which means that for a given instance in their domain, they can only take at most one value, i.e. they define a function in the algebraic sense. Properties inverse to functional properties are marked as inverse functional. Additionally, object properties can also be reflexive, symmetric, and transitive. If one likens ontology properties with algebraic relations, the semantics of these become apparent.

Similar to classes, properties can also be organised in hierarchies by means of establishing `subPropertyOf` and `superPropertyOf` relations. Methods are provided for obtaining the set of super- and sub-properties as well as the transitive closures of these sets.

The API provides methods for obtaining the names of the properties that are set for a given instance and for obtaining the sets of values for a given property and a given instance.

### A.7.2 The Ontology Resource

Ontologies in GATE are classified as language resources. In order to make use of the ontology implementation included in the main distribution, one
needs to load the ‘Ontology_Tools’ CREOLE plug-in. Once this is done, a new language resource called ‘Jena Ontology’ becomes available.

A new ontology can be created using the usual mechanism by either right-clicking on the ‘Language Resources’ sub-tree in the main resources tree and choosing ‘Jena Ontology’ or by using the ‘New Language Resource’ option in then main ‘File’ menu. This will open a dialogue like the one shown in Figure A.18.

![Figure A.18: Parameters for a new ontology](image)

When creating a new ontology, one can use an existing file to pre-populate it with data. If no such file is provided, an empty ontology is created.

A detailed description for all the parameters that are available for new ontologies follows:

**defaultNameSpace** is the base URI to be used for all new items that are only mentioned using their local name. This can safely be left empty if the ontology being loaded will only be used in read-only fashion.

**language** the default language indicator (e.g. ‘EN’ for English) for the labels and comments used throughout the ontology.

**plus one of the following**, which can be chosen from the drop-down menu:

- **owlLiteFileURL** a URL pointing to a file containing OWL Lite data
- **owlDlFileURL** a URL pointing to a file containing OWL DL data
- **owlFullFileURL** a URL pointing to a file containing OWL Full data
Once an ontology is created, additional data can be loaded that will be merged with the existing information. This can be done by right-clicking on the ontology in the resources tree and selecting ‘Load ... data’ where ‘...’ is one of the supported formats.

Another option that is available is cleaning the ontology (which will delete all the information from the ontology), and saving it to a file, which will save the data in the ontology in OWL Lite format.

A.7.3 The Ontology Viewer

The ontology support in GATE also includes a simple viewer that can be used to navigate an ontology and quickly inspect the information relating to any of the objects defined in it – classes, instances and their properties.

Figure A.19: The GATE Ontology Viewer
Figure A.19 shows the GATE ontology viewer displaying a segment of the PROTON\textsuperscript{3} ontology and the data associated with the class \texttt{Person} which is currently selected. The viewer is divided into two areas; one on the left that displays the hierarchy of classes as well as any instances, and another to the right that displays the details pertaining to the object currently selected in the hierarchical view.

The left-hand-side view displays a tree which shows all the classes defined in the ontology. The tree can have several root nodes, one for each top class in the ontology. The same tree is also used to show the instances for each of the classes. Instances that belong to several classes are shown as children of all the classes they belong to.

Whenever a item is selected in the tree view, the right-hand-side view is populated with the details that are appropriate for the selected object. If the object is an ontology class then the details include the set of direct superclasses, the set of all the superclasses using the transitive closure of the hierarchical relations, the set of direct subclasses, the set of all the subclasses, and the set of applicable properties.

When the selected object is an instance, the details displayed include the set of direct types (the list of classes this instance is known to belong to), the set of all types this instance belongs to (obtained through a transitive closure of the set of direct types), and the values for all the properties that are set.

As mentioned in the description of the data model, properties are not directly linked to the classes, but rather define their domain of applicability through a set of domain restrictions. This means that the list of properties should not really be listed as a detail for class objects, but only for instances. It is however quite useful to have an indication of the types of properties that could apply to instances of a given class. Because of the semantics of property domains, it is not possible to calculate precisely the list of applicable properties for a given class, but only an estimate of it. If a property requires its domain instances to belong to two different classes then it cannot be known with certitude whether it is applicable to either of the two classes, as it does not apply to all instances of any of those classes, but only to those

\footnote{The PROTON ontology is available from \url{http://proton.semanticweb.org/}}

152
instances the two classes have in common. Because of this, such properties will not be listed as applicable to any class.

The information listed in the details pane is organised in sub-lists according to the type of the items. Each of these lists can be collapsed or expanded by clicking on the little triangular button next to the title.

The ontology viewer is dynamic, and will update the information displayed whenever the underlying ontology is changed through the API.
Infrastructural Support for Information Extraction
Appendix B

Impact and Community Uptake

Another way of measuring the success of this work is to evaluate the impact and the acceptance by the language engineering community. While this is difficult to evaluate directly, there are several metrics that we can use:

**GATE downloads**
We have access to reliable download statistics since the GATE project started being hosted on sourceforge.net in July 2005. In order to account for seasonal variations, let us consider the three complete years for which we have statistics: 2006, 2007, and 2008. The statistics download page on sourceforge shows a total of 67,861 downloads, i.e. an annual average of 22,620, or a monthly average of 1885. Figure B.1 shows a graph representing the number of downloads on a monthly basis.

**Web site visits**
The GATE project web site\(^1\) has served a number of 6,537,316 pages between January 1\(^{st}\) 2006 and December 31\(^{st}\) 2008, meaning a monthly average of 272,388. Figure B.2 illustrates the page visits log for the web site during this period.

**Mailing list activity**
The GATE project maintains a mailing list for users’ discussions which has

\(^1\)gate.ac.uk
received 3,748 messages during the years of 2006 to 2008.

In order to evaluate the relevance of the work presented in this thesis for the GATE community, we have analysed the archives of the mailing list and counted the number of messages that mention some key-terms that are related to our work. The results of this exercise are shown in Table B.1. It is notable that one third of all email traffic on the GATE mailing list refers to JAPE and that more than half of all messages refer either directly to ANNIE or to one of its components. While the work on ontologies and machine learning receives less attention, the numbers are however not insignificant.

**Literature citations**
Another useful metric is the number of citations of this work in scientific publications. While there are a large number of papers published about GATE and ANNIE, the GATE web page dedicated to academic publications\(^2\) suggests using [Cunningham et al. 02] as the canonical reference. Google Scholar\(^3\) lists over 800 citations of this paper.

For work relating to the ontology support in GATE, the recommended citation is [Bontcheva et al. 04], which appears on Google Scholar as cited by over 100 papers.

Both of these publications include contributions by the author of this thesis. Clearly, many of the metrics presented in this section refer to the GATE project as a whole, which includes other elements beside our work on portable information extraction. However, as demonstrated by our analysis of the users’ mailing list activity, a significant fraction of the interest raised by GATE is due to the IE components provided.

---

\(^2\)http://gate.ac.uk/gate/doc/papers.html

\(^3\)http://scholar.google.com
Bibliography

[Aberdeen et al. 95]

[ACE04a]
Annotation Guidelines for Entity Detection and Tracking (EDT), Feb 2004. Available at http://www.ldc.upenn.edu/Projects/ACE/.

[ACE04b]

[ACE04c]

[Agatonovic et al. 08]

[Ahmad & Gillam 05]
K. Ahmad and L. Gillam. Automatic ontology extraction from

[Aho et al. 86] 

[Appelt & Martin 99] 

[Appelt & Onyshkevych 96] 

[Appelt 99] 

[ARPA 91] 

[ARPA 92] 

[ARPA 93] 

160
Toward Portable Information Extraction

[Bagga 98]

[Berners-Lee 99]

[Berners-Lee et al. 94]

[Bontcheva et al. 02]

[Bontcheva et al. 04]

[Brewster et al. 02]

[Brill 94]
E. Brill. Some Advances in Transformation-Based Part of Speech

[Brill 95]


[Chinchor 92]


[Ciravegna et al. 03]


[Cowie & Lehnert 96]


[Cowie et al. 93]


[Cunningham et al. 96]


[Cunningham et al. 02]

H. Cunningham, D. Maynard, K. Bontcheva, and V. Tablan. GATE:

[Day et al. 97]

[Day et al. 98]

[Dean et al. 04]

[DeJong 82]

[Dowman et al. 05]

[Finn & Kushmerick 03]
Bibliography

[Fong et al. 01]

[Frakes & Baeza-Yates 92]

[Frank et al. 05]

[Freitag & Kushmerick 00]

[Gaizauskas & Wilks 98]

[Gaizauskas et al. 95]

[Gaizauskas et al. 96]


I. Horrocks and F. van Harmelen. Reference Description of the

[Jönsson et al. 04]

[Karkaletsis et al. 99]

[Kiryakov et al. 04]

[Kogut & Holmes 01]

[Lassila & Swick 99]

[Manning & Schütze 99]
[Marcus et al. 94]

[Maynard et al. 02]

[Maynard et al. 03]

[Merchant et al. 96]

[Moldovan et al. 07]

[Oard 03]

[Popov et al. 04]
Bibliography

KIM – A semantic platform for information extraction and retrieval. 

[Sabou 04]

[Sabou et al. 05]

[Sager 78]

[Saggion et al. 02]

[SAIC 98]

[Salton & McGill 83]

[Sarawagi 08]
[Sundheim 95]

[Sundheim 98]

[Tablan et al. 03]

[Tablan et al. 06a]

[Tablan et al. 06b]

[Tablan et al. 08]
[Turmo et al. 06]

[Unicode Consortium 96]

[van Rijsbergen 79]

[Vargas-Vera et al. 07]

[Witten & Frank 99]

[Wood et al. 03]