INFORMATION EXTRACTION BY TREE AUTOMATA INFERENCE

Jury:
Prof. Dr. ir. K. De Vlamick, voorzitter
Prof. Dr. ir. M. Bruynooghe, promotor
Prof. Dr. J. Van Den Bussche, promotor
Limburgs Universitair Centrum

Prof. Dr. ir. H. Blockeel
Prof. Dr. ir. E. Duval
Prof. Dr. ir. G. Janssens
Prof. Dr. N. Kushmerick
University College Dublin
Prof. Dr. P. Adriaans
Universiteit van Amsterdam

U.D.C. 519.7    July 2003

Proefschrift voorgedragen tot het behalen van het doctoraat in de wetenschappen: informatica
door
Raymond KOSALA
Information Extraction by Tree Automata Inference

Raymond Kosala
Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 200A, B-3001 Leuven, Belgium

Abstract

The World Wide Web (Web) is a popular medium to disseminate information today. If we define accessibility of information as the ease with which the information can be obtained, then the accessibility of information on the Web leaves much to be desired. Thus, many opportunities exist to improve the current search tools on the Web. Data mining and web mining technology can play an important role in such developments.

Information extraction (IE) aims at extracting specific information from a collection of documents. From our preliminary study, we concluded that structured data, which can be in the form of trees or graphs, are involved in almost every aspect of web mining. Thus, many opportunities exist for more expressive methods in web mining.

In this thesis, we specifically explore the use of tree automata as more expressive methods for web information extraction. Existing tree automaton inference methods expect ranked trees, or trees with a fixed number of children. In contrast, HTML or XML documents have unranked tree structures. This situation opens up a new area for research in methods that infer unranked tree automata. In this thesis, we develop and apply several ranked and unranked tree automaton inference methods for IE in tree-structured documents. Our experimental results show that exploiting tree structures is indeed worthwhile. To clarify our contribution, we propose a new view on structured IE research and give a survey in that domain.
Acknowledgement

It is a great pleasure to thank all those who have made this thesis possible.

I would like to express my gratitude to my promoters Maurice Bruynooghe and Jan Van Den Bussche for their guidance during my doctorate program. I would like to thank Maurice Bruynooghe for his patience, experience, comments and suggestions that have greatly contributed to this research. In his very busy schedule, he always had the time to read what I had written and gave detailed comments. I would like to thank Jan Van Den Bussche for clarifying many theoretical aspects in my research. Actually, the idea to explore tree automata for information extraction was his. Although we collaborated long distance, his help and guidance are material to this thesis.

I would like to thank to Hendrik Blockeel for many things. First of all, he was the first person who taught me how to do research while I was working for my master’s thesis under his supervision. His guidance continued when I worked on my doctoral research.

Maurice, Jan, and Hendrik are also my closest collaborators in this thesis. They provided many invaluable inputs that helped me to improve my research.

I would like to thank the other members of my PhD committee, Erik Duval and Gerda Janssens and the members of the jury, Nicholas Kushmerick and Pieter Adriaans, who monitored my work and took effort in reading and provided me with valuable comments on earlier versions of this thesis. I would also like to thank the chairman Karel De Vlaminck.

I gratefully acknowledge financial support from the GOA Project LP+ of the K.U. Leuven, the FWO project query languages for data mining, and the Research Council scholarships programme of the K.U. Leuven.

I thank Danny De Schreye who offered me the opportunity to do research in the computer science department. I thank my colleagues and former colleagues in the growing machine learning group, and colleagues in the declarative languages and artificial intelligence group. My colleagues helped me a lot in many ways and provided a good working atmosphere. Nico Jacobs is a great colleague and many discussions with him influenced my early research work. Jan Ramon provided $\LaTeX$ templates that I use for this thesis. Nikolay Pelov and
Alexander Serebrenik helped me with procedural work. I thank the machine learning group of Leuven and Frank Neven for the collaborations. I also thank Denise Brams and Karin Michiels for their help with the administration.

I wish to thank all my friends in Leuven, especially Indonesian friends, for the help, support and for having shared many experiences.

I am deeply grateful to the love, support and caring of my father and mother; I dedicate this thesis to them. My special thanks goes to my wife Lina whose support and love enabled me to complete this work.

Last but not least, I thank God for guiding me through this journey and for making this thesis possible.

Heverlee, June 2003

Raymond Kosala
Contents

1 Introduction .......................................................... 1
   1.1 Motivation: background ...................................... 1
       1.1.1 Searching information on the Web .................. 1
       1.1.2 Improving the information access: IE and IR .... 3
   1.2 Related technologies ......................................... 4
       1.2.1 Machine learning ....................................... 5
       1.2.2 Grammatical inference ................................ 5
       1.2.3 Data mining ............................................ 6
   1.3 Motivation: specific ........................................... 7
       1.3.1 Structural information and web mining ............. 7
       1.3.2 Incorporating tree structures in structural IE ..... 8
   1.4 Structure of the work and contributions .................. 10
   1.5 Bibliographical note .......................................... 11

2 Basic definitions and concepts .................................... 13
   2.1 Introduction .................................................. 13
   2.2 Alphabets, strings, and languages .......................... 14
   2.3 Graphs and trees ............................................. 15
   2.4 Regular expressions and finite automata ................... 16
   2.5 Context free grammars ....................................... 19
   2.6 Tree automata ................................................ 20
       2.6.1 Ranked tree automata .................................. 20
       2.6.2 Unranked tree automata ............................... 21
   2.7 Languages, grammars and automata .......................... 24
       2.7.1 Relationship between formal languages ............ 24
       2.7.2 The Chomsky hierarchy ................................ 25
   2.8 Structured documents ......................................... 26
       2.8.1 Three types of documents ............................... 26
       2.8.2 HTML .................................................... 29
       2.8.3 XML ...................................................... 32
       2.8.4 Document schema ....................................... 32
3 Information extraction and grammatical inference 41
  3.1 Introduction ........................................ 41
  3.2 Information extraction (IE) .......................... 42
     3.2.1 Definition ........................................ 42
     3.2.2 Applications of IE ............................... 43
     3.2.3 Taxonomy of IE ................................. 44
     3.2.4 Classical or traditional IE ..................... 45
     3.2.5 IE from semi-structured data .................. 47
     3.2.6 IE from structured data ....................... 48
     3.2.7 Discussion on IE systems .................... 49
  3.3 Grammatical inference ............................... 50
     3.3.1 Introduction ........................................ 50
     3.3.2 Applications of grammatical inference .......... 51
     3.3.3 Learning from positive examples .............. 52
     3.3.4 Inference of regular grammars .................. 53
        3.3.4.1 Inference of k-reversible languages ........ 54
        3.3.4.2 Inference of k-contextual languages ........ 55
        3.3.4.3 Inference of k-testable languages ........... 56
        3.3.4.4 Examples of regular grammar inference ....... 57
     3.3.5 Inference of tree grammars ..................... 59
        3.3.5.1 Inference of k-reversible tree automata ...... 60
        3.3.5.2 Inference of k-testable tree automata ....... 63
        3.3.5.3 Inference of probabilistic tree automata .... 64
        3.3.5.4 Inference of probabilistic k-testable tree automata 65
        3.3.5.5 Inference of stochastic deterministic tree automata .......... 66
        3.3.5.6 Inference of stochastic many-sorted tree automata 66
        3.3.5.7 Inference of stochastic ranked node rewriting grammars .......... 67
        3.3.5.8 GIFT: inference of logical terms ............. 68
  3.4 Summary ............................................. 69

4 IE by tree automaton inference 71
  4.1 Introduction ........................................ 71
  4.2 From string-based to tree-based IE .................. 72
     4.2.1 String-based IE .................................... 72
     4.2.2 Tree-based IE .................................... 72
  4.3 IE by grammatical inference .......................... 74
  4.4 Approach and issues ................................. 75
5.6 The advantages and limitations of our methods .................. 128
5.7 Summary ......................................................... 130

6 Related work .................................................................. 133
  6.1 Introduction ............................................................ 133
  6.2 Information extraction: an emerging field of research .......... 134
  6.3 IE from structured data ............................................. 136
    6.3.1 Manually-built structured systems .......................... 139
    6.3.1.1 String-based ................................................. 139
    6.3.1.2 Tree-based .................................................. 140
    6.3.2 Semi automatically-built structured systems .............. 142
      6.3.2.1 String-based .............................................. 142
      6.3.2.2 Tree-based ............................................... 144
    6.3.3 Automatically-built structured systems ................... 145
      6.3.3.1 String-based .............................................. 145
      6.3.3.2 Tree-based ............................................... 146
  6.4 Other related work .................................................. 147
    6.4.1 Wrapper maintenance .......................................... 147
    6.4.2 Schema / DTD inference ...................................... 150
    6.4.3 Xpath and query languages ................................... 151
    6.4.4 Information integration ...................................... 153
  6.5 Structured IE research: the current state ........................ 155
    6.5.1 Towards multiple representation ............................ 155
    6.5.2 No free lunch .................................................. 159
  6.6 Which structured IE method? ....................................... 162
  6.7 Conclusion .......................................................... 165

7 Conclusion ...................................................................... 167
  7.1 Summary .............................................................. 167
  7.2 Further work .......................................................... 171
    7.2.1 Extraction of parts of the text node ....................... 171
    7.2.2 Multi-slot extraction and extraction efficiency ........... 172
    7.2.3 Alternative methods .......................................... 173

A Examples of the induced tree automaton ............................ 175
  A.1 Inference on ranked tree ........................................... 175
  A.2 Inference on unranked tree ........................................ 180
List of Figures

1.1 Difference between IE and IR ..................................... 4

2.1 The digraph (\{1, 2, 3\}, \{i \rightarrow j| i < j\}) ................. 15
2.2 An example of a tree ................................................. 16
2.3 An example of DFA ................................................ 18
2.4 An example of PTA ................................................ 19
2.5 A ranked tree .................................................... 20
2.6 A tree and an accepting run of the automaton in Example 2.3 . 21
2.7 A tree and a rejecting run of the automaton in Example 2.3 . 22
2.8 An unranked tree .................................................. 22
2.9 A tree and an accepting run of the automaton in Example 2.4 . 23
2.10 A tree and a rejecting run of the automaton in Example 2.4 . 23
2.11 Tree examples .................................................... 25
2.12 An example of unstructured document taken from Reuters-21578 dataset (www.research.att.com/lewis) ....................... 27
2.13 Two examples of semi-structured document taken from the DBWorld mailing list (www.cs.wisc.edu/dbworld) and LA Times classifieds (www.latimes.com) ................................. 28
2.14 An example of structured document taken from OKRA dataset (www.isi.edu/~museaRISE). The upper part is the rendered view of the HTML document from a web browser, while the lower part is (a part of) the HTML document. .......... 30
2.15 An example of a HTML file ........................................ 31
2.16 A screen grab of a browser displaying the HTML file of Figure 2.15 31
2.17 An XML file .................................................... 32
2.18 A DTD of an XML file .......................................... 33
2.19 Two examples of a token level representation .................... 35
2.20 Two examples of a tree level representation ..................... 36
2.21 An example that gives rise to visual and geometric level represen-tations ............................................. 36
### LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>IE taxonomy</td>
<td>46</td>
</tr>
<tr>
<td>3.2</td>
<td>An example of context-free grammar that produce $L = 0^*1^n$.</td>
<td>50</td>
</tr>
<tr>
<td>3.3</td>
<td>A PTA for example 3.1</td>
<td>58</td>
</tr>
<tr>
<td>3.4</td>
<td>A PTA for example 3.2</td>
<td>58</td>
</tr>
<tr>
<td>4.1</td>
<td>From string-based to tree-based IE</td>
<td>73</td>
</tr>
<tr>
<td>4.2</td>
<td>An example of an HTML document</td>
<td>77</td>
</tr>
<tr>
<td>4.3</td>
<td>An example of partial HTML tree of Figure 4.2</td>
<td>78</td>
</tr>
<tr>
<td>4.4</td>
<td>An example of conversion from unranked into binary tree.</td>
<td>83</td>
</tr>
<tr>
<td>4.5</td>
<td>Encoding steps</td>
<td>83</td>
</tr>
<tr>
<td>4.6</td>
<td>Generalizing a fork</td>
<td>86</td>
</tr>
<tr>
<td>4.7</td>
<td>A fork and its two possible generalizations</td>
<td>89</td>
</tr>
<tr>
<td>4.8</td>
<td>An example and the effect of convert label</td>
<td>94</td>
</tr>
<tr>
<td>5.1</td>
<td>A DTD of the small Shakespeare dataset</td>
<td>103</td>
</tr>
<tr>
<td>5.2</td>
<td>The graphs of F1 score vs $k$</td>
<td>107</td>
</tr>
<tr>
<td>5.3</td>
<td>The graphs of training time vs $k$</td>
<td>108</td>
</tr>
<tr>
<td>5.4</td>
<td>The graph of extraction time vs $k$</td>
<td>109</td>
</tr>
<tr>
<td>5.5</td>
<td>The graphs of the number of states vs $k$</td>
<td>110</td>
</tr>
<tr>
<td>5.6</td>
<td>A DTD of the full Shakespeare dataset</td>
<td>116</td>
</tr>
<tr>
<td>5.7</td>
<td>The learning curve of the unranked algorithm</td>
<td>118</td>
</tr>
<tr>
<td>5.8</td>
<td>The performance of the algorithms with noisy examples</td>
<td>126</td>
</tr>
<tr>
<td>5.9</td>
<td>Two trees and their corresponding binary trees</td>
<td>127</td>
</tr>
<tr>
<td>6.1</td>
<td>An XML document</td>
<td>151</td>
</tr>
<tr>
<td>6.2</td>
<td>An example of string-based first-order extraction rule</td>
<td>157</td>
</tr>
<tr>
<td>6.3</td>
<td>An example of tree-based first-order extraction rule</td>
<td>158</td>
</tr>
<tr>
<td>6.4</td>
<td>The DTD of our example</td>
<td>159</td>
</tr>
<tr>
<td>6.5</td>
<td>A simplified XML tree produced from the DTD of Figure 6.4</td>
<td>161</td>
</tr>
<tr>
<td>6.6</td>
<td>Some characteristics of structured IE systems</td>
<td>164</td>
</tr>
<tr>
<td>7.1</td>
<td>An example of partial HTML tree</td>
<td>171</td>
</tr>
<tr>
<td>7.2</td>
<td>An example of partial extended HTML tree</td>
<td>172</td>
</tr>
</tbody>
</table>
List of Tables

2.1 Equivalence between languages, grammars, and automata . . . 24
3.1 Gold’s result of the learnability of language classes . . . . 53
3.2 Some upgrades to tree automaton inference algorithms . . . . 61
3.3 Some characteristics of tree automaton inference algorithms . 62
5.1 Comparison of the results . . . . . . . . . . . . . . . . . . . 105
5.2 Parameters used for the experiments . . . . . . . . . . . . . 105
5.3 The number of examples needed for good generalization . . . 111
5.4 The test on the rest of the larger datasets . . . . . . . . . . . 111
5.5 Comparison to the other IE systems on the larger datasets . . 112
5.6 Comparison of the results . . . . . . . . . . . . . . . . . . . 115
5.7 Parameters used for the experiments . . . . . . . . . . . . . 115
5.8 The number of examples and training time per example needed for algorithm 6 . . . . . . . . . . . . . . . . . . . . . . . . . . . . 117
5.9 Sample complexity . . . . . . . . . . . . . . . . . . . . . . . 120
5.10 Sample complexity of the Bigbook-name task . . . . . . . . . 120
5.11 Sample complexity of the Bigbook-address task . . . . . . . . 121
5.12 Results on the Syllabus dataset . . . . . . . . . . . . . . . . 123
5.13 Results of the gl-testable algorithm without distinguishing context124
5.14 Results in the small Shakespeare dataset with noisy examples . 125
6.1 Some examples of IE systems for structured data . . . . . . . 137
6.2 Ability of systems to carry out the four extraction tasks . . . 162
List of Symbols

An overview of some symbols and notations that are used in the text.

\[
\begin{align*}
\varepsilon & \quad \text{element} \\
\not\in & \quad \text{not element of} \\
\subseteq & \quad \text{subset} \\
\subset & \quad \text{proper subset} \\
\cup & \quad \text{union} \\
\cap & \quad \text{intersection} \\
\emptyset & \quad \text{emptyset} \\
\approx & \quad \text{approximately} \\
|A| & \quad \text{the number of elements in set } A \\
2^A & \quad \text{the set of all subsets of } A \text{ or the power set of } A \\
\exists & \quad \text{exists (existential quantifier)} \\
\forall & \quad \text{for all (universal quantifier)} \\
| & \quad \text{such that} \\
* & \quad \text{a wildcard that matches any symbol} \\
\Sigma & \quad \text{a finite alphabet} \\
\Sigma^* & \quad \text{the infinite set of strings made up of zero or more letters from } \Sigma \\
\# & \quad \text{a symbol that is not in } \Sigma \\
\epsilon & \quad \text{the empty string or error rate} \\
\delta & \quad \text{the transition function or the probability that a learner fails} \\
\text{to learn the target concept} \\
|u| & \quad \text{the length of string } u \\
uv & \quad \text{the concatenation of the strings } u \text{ and } v \\
L & \quad \text{a language } L \text{ is any subset of } \Sigma^* \\
M & \quad \text{an automaton, which can be a string or a tree automaton} \\
L(M) & \quad \text{the language } L(M) \text{ recognized by automaton } M \\
G & \quad \text{a grammar} \\
P_r(L) & \quad \text{the prefixes of elements of } L \\
T_L(w) & \quad \text{the set of strings that can follow string } w \text{ in } L
\end{align*}
\]
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$V$</td>
<td>the set of all ranked or unranked labels</td>
</tr>
<tr>
<td>$</td>
<td>V</td>
</tr>
<tr>
<td>$T_v$</td>
<td>the set of all ranked or unranked trees built with labels from $V$</td>
</tr>
<tr>
<td>$t$</td>
<td>a tree</td>
</tr>
<tr>
<td>$k$ or $k_c$</td>
<td>a natural number</td>
</tr>
<tr>
<td>$\mathcal{R}$ or $FS$</td>
<td>the set of all $(k - 1)$-roots from a set of trees $T$, $r_{k-1}(T)$</td>
</tr>
<tr>
<td>$\mathcal{F}$</td>
<td>the set of all $k$-forks from a set of trees $T$, $f_k(T)$</td>
</tr>
<tr>
<td>$\mathcal{S}$</td>
<td>the set of all $(k - 1)$-subtrees from a set of trees $T$, $s_{k-1}(T)$</td>
</tr>
<tr>
<td>$Prec$ or $P$</td>
<td>precision</td>
</tr>
<tr>
<td>$Rec$ or $R$</td>
<td>recall</td>
</tr>
<tr>
<td>$F1$</td>
<td>F1 score</td>
</tr>
<tr>
<td>$h$</td>
<td>a hypothesis</td>
</tr>
<tr>
<td>$H$</td>
<td>a set of hypotheses</td>
</tr>
<tr>
<td>$x$</td>
<td>a special label for labeling the target fields</td>
</tr>
<tr>
<td>$v_x$</td>
<td>labels that are marked with suffix $x$</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

This chapter aims at giving an introduction to the work presented in this thesis. In the first section, we present a general motivation of our work. We start with describing the problem of searching for information on the Web and indicating some opportunities to improve the current search tools on the Web. Then, we argue that information extraction (IE) technology could be used to improve the intelligence of current search engines.

After that, we briefly describe the field of machine learning, followed by descriptions of grammatical inference and data mining.

In the next section, we repeat the conclusion from our initial study on web mining (Kosala and Blockeel 2000), which is the main motivation for our further research in this thesis. Next, we elaborate on our motivations and describe in more detail the characteristics of our method.

Finally, we describe the structure of this thesis and highlight our main contributions.

1.1 Motivation: background

1.1.1 Searching information on the Web

In the early nineties, the term “World Wide Web” (WWW), which refers to a rapidly growing network of computers that are connected to each other via the Internet, started to become popular. While the Internet itself was around long before that time, it is only in the early 90’s that the concept of the Web as a single, huge and widely-distributed repository of information came into existence.

Unfortunately, there is a drawback to the Web as an information repository. While, in principle, the amount of information that is available is huge, in
practice it may be much less, in the sense that it may not be obvious how to find the relevant information. If we define accessibility of information as the ease with which the information can be obtained (i.e., how much knowledge or expertise is needed to obtain it), then the accessibility of information on the Web leaves much to be desired. The knowledge and expertise required to get certain information from the Web (in addition to being able to use a web browser) may consist of:

- knowledge of the URL (Universal Resource Locator) where the information resides, or

- knowledge concerning where to find and how to use a search engine on the Web; the use of a search engine may range from very simple queries (entering a keyword) to rather complex queries (involving Boolean operators, the specification of a domain name, ...), or

- knowledge of how to write a special-purpose agent that searches the Web for the information.

It is easy to find information if one knows where it is, but it is unreasonable to assume that the user always has this information. On the other hand, the third option (writing a special-purpose program) presupposes a special skill that very few users have. Hence, the availability of good and easy-to-use search engines is crucial for increasing the accessibility of information on the Web. The reason why we consider the information on the Web to be poorly accessible is that the quality of current-day search engines still leaves much room for improvement.

Certain kinds of information can easily be found by using a search engine. E.g., if the user types in “Amsterdam” as a keyword to search for, chances are very high that the first pages the search engine returns contain information about the city of Amsterdam. But if he or she wants to learn about computers, typing in “computers” may give millions of pages, only very few of which will contain relevant information. Typing in a phrase such as “learn about computers” will probably help, but may still return too many pages, the majority of which are of little interest; moreover, the most interesting ones may be so far down the list that the user will never discover them, and some very interesting pages may not even be in the list because they do not contain the keywords themselves but only related words.

At this point it is useful to introduce the terms recall and precision which are often used in the domains of information retrieval and information extraction to indicate the quality of the result of a search. In the context of the above example, recall refers to the proportion of the actually interesting pages that the search engine returns, and precision refers to the proportion of pages returned by the search engine that are actually interesting. More formally: given a set
1.1. MOTIVATION: BACKGROUND

S (e.g. the set of all web pages) from which we want to extract the set of all members of S that satisfy some criterion C (denoted \( S(C) \)), the result of the extraction process is a set \( A \), and that the result of the extraction process that satisfy some criterion \( C \) is a set \( A(C) \), then the recall is defined as \( R = |A(C)|/|S(C)| \) and the precision as \( P = |A(C)|/|A| \).

In the above example, \( S \) is the set of all web pages on the Web; however in other contexts it could be the set of all websites (where a site is defined as a collection of pages that all belong together), the set of all images occurring on the Web, the set of all words occurring on the Web, the set of all words occurring in a given document, etc.

Obviously, when looking for information on the Web, the user prefers answer sets with high recall and precision; ideally the system should return everything that is of interest, and nothing else. Present day web technology is limited in the sense that for many kinds of questions, it is very hard to formulate a question in such a way that a set of answers with high recall and precision is returned. Actually, recent research (Lawrence and Giles 1999) has shown that a large percentage of the Web is not even indexed, which further reduces the maximum possible recall of search engines. The problem of indexing is beyond the scope of this text; here we focus on obtaining information from those parts of the Web that are indexed.

1.1.2 Improving the information access: IE and IR

In order to improve the accessibility of the information on the Web, there is a need for more intelligent search engines. How to build systems that return relevant information from a large information repository is currently the subject of several research domains. We here distinguish two research domains:

- Information retrieval (IR): deals with the problem of how to find relevant documents. This problem is closely related to the problem of finding web pages, as discussed above. A well-known example of an IR system is the Internet search engine.

- Information extraction (IE): deals with the problem of how to extract certain information from a single document. For instance, assuming the document is an article published on the Web, find the name of the author of the article; or find the names of all authors mentioned in the reference section of the article.

For further clarification on the difference between the IE and IR research domains, see Figure 1.1. These research domains are linked to each other by the technology they use, and, in the same way, they are linked to knowledge discovery and data mining. Indeed, techniques from data mining can be used (to support the goals of information retrieval and information extraction) as we
shall see later on. However, because of its ability to construct new knowledge, data mining (or the knowledge discovery process as a whole) can also be used for yet another task: extending the Web with new information. This information could itself be made public on the Web, or it could serve the private purposes of the user (e.g., to support knowledge-based inference and problem solving). When knowledge discovery is a goal in itself, then information retrieval and extraction becomes a subprocess of the knowledge discovery, because standard data mining techniques usually cannot work with such heterogeneous information as is found on the Web, and preprocessing the data may involve IE and IR technology.

In summary, there is a complex interaction between data mining, and information retrieval and information extraction: both may employ the other to achieve their goals. A data mining technique might use data that have been extracted from the Web using IR and IE, and the latter may again use (other) data mining techniques.

Thus, as argued in (Kosala and Blockeel 2000), we could improve the intelligence of search engines by improving the IR and IE tasks on the Web with the help of web mining technology.

1.2 Related technologies

This thesis will present the application of machine learning and related techniques to the problem of information extraction from structured documents found on the Web, such as HTML and XML documents. Below we give a brief introduction to these related techniques.
1.2. RELATED TECHNOLOGIES

1.2.1 Machine learning

Learning is an activity that is experienced by almost all of the living creatures in our world. As human beings we almost certainly learn something new everyday. Learning is something that is almost inseparable from the experience of being alive. Wise people tell us that we have to learn if we want to become an “intelligent” person. With learning, we improve ourselves through experience. Without the learning capability, it is almost certain that we would not exhibit intelligent behavior.

This is one reason why machine learning (ML) is at the heart of the artificial intelligence (AI) research field. The field of machine learning was invented with the goal, among others, of making machines or computer programs able to mimic how living creatures, especially humans, learn. According to Mitchell (Mitchell 1997), the field of machine learning is concerned with constructing computer programs that can automatically improve with experience. Machine learning is an interdisciplinary field that originates from AI, statistics, biology, cognitive science, computational complexity, information theory, etc.

In machine learning, we can distinguish two major types of learning that are related to the methods described in this thesis, namely, supervised and unsupervised learning.

In supervised learning, the learner receives classified training examples. This means that each of the examples has a class associated with it. A teacher or oracle typically provides this class information. Then the learner’s task is to find a model that can be used to classify most of the training examples correctly. The learned model is also expected to be able to classify unseen examples correctly. Examples of the learning tasks in this type are concept learning and categorization.

In unsupervised learning, the learner receives unclassified training examples. The learner has to find patterns that describe the training examples. Examples of the learning tasks that can be categorized as this type of learning are clustering and discovery of frequent patterns.

1.2.2 Grammatical inference

Grammatical inference refers to the process of learning rules from a set of labeled examples. It is a subfield of machine learning and belongs to a class of inductive inference problems (Angluin and Smith 1983). In the literature, it is also often referred to as automata induction, grammar induction, or automatic language acquisition. It is a well-established research field in AI that goes back to Gold’s work (Gold 1967).

Typically the target domain of a grammatical inference algorithm is a class of formal languages (a set of strings over some alphabet $\Sigma$) and the hypothesis space is a family of grammars. A grammatical inference process can be seen as
a search problem through the set of possible grammars according to a certain quality criterion.

Many different classifications of grammatical inference algorithms have been made, see for instance (Fu and Booth 1975a; Fu and Booth 1975b; Gonzalez and Thomason 1978; Anghin and Smith 1983; Sakakibara 1997; Parekh and Honavar 1998). For the purpose of this thesis, it is useful to classify them according to their target domain. One can distinguish:

- String or regular grammar inference, where the target domain is a class of string or regular languages.
- Context-free grammar inference, where the target domain is a class of context-free languages.
- Tree grammar inference, where the target domain is a class of tree languages.

This thesis develops and applies some grammatical inference techniques that can be categorized as tree grammar (or tree automaton) inference algorithms.

We can classify grammatical inference algorithms that learn a class of grammars from positive and negative examples as supervised learning. On the other hand, we can classify grammatical inference algorithms that learn a class of grammars from positive examples only, where these examples are unlabeled, as unsupervised learning.

The learning task that we consider in this thesis uses only unlabeled documents as positive examples and uses no negative examples. However, our learning task can be classified as supervised learning because a teacher has to label the parts of the document where the fields that will be extracted are located.

1.2.3 Data mining

Data mining concerns the automatic discovery of patterns or models from data. The idea is to apply computer programs that sift through databases automatically to find regularities or patterns. The discovered pattern or knowledge can then be used for the prediction and categorization of new data, or just to give a general description of the data. Data mining is a part and sub-process of knowledge discovery in databases (KDD). KDD refers to the overall process of discovering potentially useful and previously unknown information or knowledge from the databases (Adriaans and Zantinge 1996; Fayyad, Piatetsky-Shapiro, and Smyth 1996a). In the literature, however, the term data mining is sometimes also used to denote KDD.

If we follow the definition of KDD from (Fayyad, Piatetsky-Shapiro, and Smyth 1996a), then there are some additional processes, besides data mining,
in KDD. These additional processes are non-trivial. KDD more or less consist of the following iterative process:

Data collection → Data cleaning and preprocessing → (Data mining) → Interpretation of data mining results.

They are non-trivial because these processes typically require manual intervention and specific knowledge from domain experts. While the data mining process can be largely automated, these additional processes are mostly done manually by domain experts, hence they are the most time consuming part of the whole KDD process.

There is a close relationship between data mining, machine learning and advanced data analysis (Mitchell 1999). One of the characteristics of KDD is that KDD aims to discover patterns or knowledge that is “understandable” or that can be interpreted by humans.

1.3 Motivation: specific

1.3.1 Structural information and web mining

In (Kosala and Blockeel 2000) we surveyed the applications of data mining on the Web and showed that data mining techniques have been successfully used in several aspects of web mining. Basically, web mining is the use of data mining techniques to discover and extract information from Web documents and services automatically (Etzioni 1996). Web mining refers to the overall process of discovering potentially useful and previously unknown information or knowledge in web data.

We have divided web mining research into three categories. They are: web content mining, web structure mining, and web usage mining. Web content mining describes the discovery of useful information from Web contents or data or documents. Web structure mining tries to discover the model underlying the link structures of the Web. Finally, web usage mining tries to make sense of the data generated by the web surfer’s sessions or behaviors.

We conclude that structured data, which can be in the form of trees or graphs, are involved in almost every aspect of web mining. Many of the data mining techniques that we survey are either extensions of old techniques that make them more expressive, or new techniques that are more expressive than the old ones. The additional expressiveness is needed to work with web data because the Web offers data that is quite different from traditionally flat data. Thus, many opportunities exist for employing both old and new data mining techniques while working with structured data.

This conclusion provides a motivation for the work which is described in this thesis. Specifically, the main goals of this thesis are as follows:
• to exploit structural information, such as the tree structure of the document, for web information extraction, a web mining application, and

• to give experimental evidence that the incorporation of the structural information that is available in web data is beneficial for structural information extraction.

This thesis focuses on the application of those machine learning methods that can be categorized as grammatical inference techniques, to extract information from web content data. In a sense, our work fits within the category of web content mining. However, we actually perform only a partial web content mining. This is because the thesis assumes that the web content data are already retrieved and collected: we do not deal with the IR problem of retrieving these web documents. However, we do perform two simple preprocessing steps, namely data cleaning and labeling the target fields in the datasets. The data cleaning step, which is described in Section 2.8.2, is usually needed to fix syntactical errors in HTML documents.

This thesis focuses on IE from structured documents such as HTML and XML documents. This work is different from traditional work in IE, which uses unstructured texts. A notable difference is that our methods do not exploit any linguistic knowledge, even though linguistic knowledge is used intensively in traditional IE work. Instead, our methods exploit the structure of the information to be extracted and the context in which it occurs.

We apply and experiment with some existing tree grammar inference algorithms, develop some extensions of the $k$-testable tree automaton inference algorithm, and also develop our own algorithm for unranked trees.

1.3.2 Incorporating tree structures in structural IE

An IE system that was intended for structured documents and does not use linguistic knowledge is called a wrapper. Some structural IE systems that have been applied to work with structured data are (Hsu and Dung 1998; Freitag and McCallum 1999; Kushmerick, Weld, and Doorenbos 1997; Kushmerick 2000a; Muslea, Minton, and Knoblock 2001; Freitag and Kushmerick 2000; Soderland 1999; Chidlovskii, Ragelii, and de Rijke 2000). These IE systems can be seen as using machine learning or grammatical inference techniques to induce a kind of delimiter-based string pattern. These methods, however, consider the structured document to be a string, not a tree.

Although it is relatively easy and efficient to learn and parse the string-based patterns for IE, there is a problem with string-based patterns when extracting target fields, whose locations in the document depend on some structural context. A structural context that is close to the target field in the tree structure of the document can become arbitrarily far away in the string representation.
of the document. This makes the learning task very difficult for string-based methods, and results in wrappers with rather poor performance. As we shall see, many string-based extraction methods have difficulties in this situation.

Structured documents such as HTML and XML documents, however, have a tree structure. Thus, it is a natural extension to consider utilizing the tree structure to extract structured documents, and to explore the use of tree automata for IE from structured documents. Indeed, tree automata are well-established and natural tools for processing trees (Comon, Dauchet, Gilleron, Jacquemard, Lugiez, Tison, and Tommasi 1999). An advantage to using the more expressive tree formalism is that the extracted field can depend on its structural context in a document, a context that is lost if the document is linearized into a string.

Actually, the idea of utilizing the (partial or simplified) tree structure of structured documents is not new. There have been some previous works that follow this idea. Chapter 4 and Chapter 6 contain more discussion about this. However, the use of tree automata to extract information from web documents is a new idea.

In this thesis, we develop and apply some tree automaton inference algorithms to extract information from tree structured web documents. We focus on grammatical inference methods that are expressive, efficient and easy to apply. These three points can be elaborated as follows:

- Our methods are expressive in the sense that they utilize tree automata to process tree-structured data. Our methods are more expressive than most existing IE systems, because these systems use a method based on string automata inference.

- Our methods are efficient in the sense that the time needed by the algorithms to learn the extractor is short. Actually, some of our algorithms can learn more quickly than some string-based methods.

- Our methods are easy to apply and require minimal effort from the user in the training process. The user only needs to label a relatively small number of examples to train the extractor. In contrast, some other methods might need preprocessing before the algorithm can learn the extractor. Some examples of these preprocessing steps that are usually done for HTML documents are: splitting the document into small fragments and selecting some of them for use as training examples, specifying manually the length of a window for the prefix, suffix and target fragments, and dealing with special tokens or landmarks such as ‘>’ or ‘;’.

We have developed and implemented some prototypes of IE systems based on several tree automaton inference algorithms. In the experiments, we consider
the performance, learnability and efficiency of some of our methods. These three points can be explained as follows:

- The performance of our methods is evaluated based on precision and recall, and a close comparison to current state-of-the-art methods is made.
- The learnability of our methods is analyzed using criteria from the computational learning theory field: identification in the limit and the probably approximately correct (PAC) framework.
- The complexity is analyzed and empirical running times are provided to show the efficiency of our methods.

In this thesis, we will not focus on the definition and usage of some terms. We will use the commonly used terms in our research domain. For example, we will not differentiate between the terms “information extraction” and “data extraction”. Similarly, from the KDD point of view, the terms “information” and “knowledge” are interchangeable (Fayyad, Piatetsky-Shapiro, and Smyth 1996b).

1.4 Structure of the work and contributions

This dissertation deals with the problem of information extraction from structured web documents such as HTML and XML documents. Our solution is to apply tree automata as extractors for tree-structured documents. It has been argued in several studies that the manual coding of extraction rules would create a bottleneck in the development process and would not be feasible for a dynamic medium such as the Web. For this reason, we develop machine learning techniques that infer tree automata from labeled examples automatically, rather than relying on the manual coding of tree automata.

This chapter situates the work and explained its interdisciplinary nature. The rest of this thesis is structured as follows:

Chapter 2. Presents some basic concepts that are used in the thesis. These concepts range from definitions taken from formal language theory, finite automata and tree automata, to a brief introduction to structured documents, such as HTML and XML documents, and to computational learning theory.

Chapter 3. Gives a general overview and a taxonomy of the IE domain, which consists of IE from unstructured texts, IE from semi-structured data, and IE from structured data. This chapter also introduces the domain of grammar inference, and surveys some regular and tree grammar inference methods.
Chapter 4. Presents a framework of two approaches to tree-based IE, which is shown in Figure 4.1, that distinguish them from the string-based IE. Then we describe how an IE problem can be cast as a tree grammar inference problem. Finally, this chapter proposes several tree automaton inference algorithms and describes some of the issues involved.

Chapter 5. Demonstrates the performance, learnability, and efficiency of our methods. We conclude the chapter with an assessment of the advantages and limitations of our methods.

Chapter 6. Gives a survey of the related structured IE work and the current state of structured IE research. Finally, the chapter analyses the practical issues involved in selecting which method to use.

Chapter 7. Summarizes the thesis and proposes some future work.

The main contributions of this dissertation fall into the following areas.

- Idea:
  - In general: investigating the need for more expressive methods in web mining.
  - In particular: exploring tree automata as more expressive methods for web information extraction.

- Information extraction community:
  - Proposing the use of a novel method based on tree automata for information extraction in tree-structured documents, and showing that this method is indeed worthwhile.
  - Proposing a new view on and survey of structured IE research.

- Grammatical inference community:
  - Developing several efficient tree grammar inference methods for information extraction applications.

1.5 Bibliographical note

Parts of this thesis have been published in the literature. Below is a list of the key papers that are integrated in this thesis.

– Used in Chapter 1: motivation.


  – Used in Chapter 1: motivation, Chapter 2: structured documents, and Chapter 6: information integration.


  – Used in Chapter 4: \( k \)-testable method, and Chapter 5: test on the benchmark datasets.


  – Used in Chapter 4: \( g \)-testable method, and Chapter 5: test on the benchmark datasets.


  – Used in Chapter 4: \( gl \)-testable method, and Chapter 5: experiments with ranked tree automata.


  – Used in Chapter 4: unranked method, and Chapter 5: experiments on benchmark datasets.
Chapter 2

Basic definitions and concepts

2.1 Introduction

In this chapter we present some basic concepts that provide a background for the thesis.

In the first sections, we introduce some basic definitions from formal language theory. We first introduce some brief definitions of alphabets, strings, languages, graphs, and trees. Then, we give the definition of regular expressions and its corresponding representation in finite automata. In the next section, we introduce automata that are more expressive than finite automata: tree automata. Specifically we describe two types of tree automata: ranked and unranked. Next, we describe the correspondence between some classes of languages, grammars, and automata, and the Chomsky hierarchy that can be formed. Then, we describe types of structured documents, such as HTML and XML documents, preprocessing for HTML documents, document schema and DTD, and some possible representations of structured documents. Finally, we briefly describe the computational learning theory field.

The background on formal language theory is mostly based on (Hopcroft and Ullman 1979; Angluin 1982); the background on tree automata is based on the following papers: (Comon, Dauchet, Gilleron, Jacquet-Lagrèze, Lugiez, Tison, and Tommasi 1999; Murata 2000; Brüggemann-Klein, Murata, and Wood 2001) and a more readable introductory paper (Neven 2002); and the background on markup languages is mostly based on (Kosala, Blockeel, and Neven 2002).
2.2 Alphabets, strings, and languages

A symbol is any single object such as a letter or digit. An alphabet \( \Sigma \) is a finite, non-empty set of symbols. For example, the binary alphabet is \( \{0,1\} \) and the first four letters of the lowercase English alphabet is \( \{a,b,c,d\} \).

A string or sentence is a finite sequence of zero or more symbols over an alphabet. For example, \( a, b, \) and \( c \) are symbols and \( abc \) is a string. The length (or cardinality) of a string \( w \), denoted \( |w| \), is defined as the number of symbols in the string. For example, \( |1001| = 4 \). The empty string, denoted by \( \epsilon \), is the string containing zero symbols. Thus \( |\epsilon| = 0 \). If \( u \) and \( v \) are strings over an alphabet \( \Sigma \), then the concatenation of \( u \) and \( v \), denoted \( uv \) or \( uw \), is also defined as a string over \( \Sigma \). The string \( v \) is a prefix of a string \( w \), \( Pr(w) \), if and only if there exists a string \( u \) such that \( vu = w \).

If \( u \) is a string and \( i \) is an integer, then the notation \( u^i \) denotes the string obtained by concatenating \( i \) copies of the string \( u \). Similarly, if \( S \) is a set of symbols, the \( S^i = \{ u_1 \ldots u_i \mid u_j \in S, 1 \leq j \leq i \} \). Thus, \( \Sigma^i \) denotes the set of all strings over \( \Sigma \) with length \( i \). The set \( \Sigma^0 \) contains the empty string \( \epsilon \). The set of all strings over an alphabet \( \Sigma \) is denoted by \( \Sigma^* \). Clearly, \( \Sigma^* \) is an infinite set.

A (formal) language is a set of strings of symbols from an alphabet. Therefore, any language \( L \) defined over an alphabet \( \Sigma \) is a subset of \( \Sigma^* \). Both the empty set, \( \emptyset \), and the set consisting of the empty string \( \{\epsilon\} \) are languages. Note that they are distinct; the latter has a member while the former does not. Another example is the language over the alphabet \( \{0,1\} \), where strings have length less than or equal to 2: \( \{0,1\} \leq 2 = \{\epsilon,0,1,00,01,10,11\} \).

The concatenation of two languages \( L_1 \) and \( L_2 \) is defined to be the language

\[ L_1L_2 = \{ uv \mid u \in L_1, v \in L_2 \} \]

In addition, \( L^i = \{ u_1 \ldots u_i \mid u_j \in L, 1 \leq j \leq i \} \). The closure \( L^* \) of a language \( L \) is defined to be the language

\[ L^* = \sum_{i=0}^{\infty} L^i, \]

where \( L^0 \) contains the empty string \( \epsilon \) only. For example, if \( L = \{x\} \) then \( L^* = \{\epsilon, x, xx, xxx, \ldots\} \). If \( L \) is a language, the set \( Pr(L) \) of prefixes of elements of \( L \) is defined by

\[ Pr(L) = \{ u \mid \text{for some } v, uv \in L \}. \]

The set of strings that can follow string \( w \) in a language \( L \) is denoted as

\[ T_L(w) = \{ v \mid vw \in L \}. \]
2.3 Graphs and trees

A graph, denoted $G = (V, E)$, consists of two things: a finite set of vertices (or nodes) $V$ and a set of unordered pairs of vertices $E$ called edges. A directed graph (or digraph), also denoted $G = (V, E)$, consists of: a finite set of vertices (or nodes) $V$ and a set of ordered pairs of vertices $E$ called arcs or directed edges or simply edges. An example of digraph is shown in Figure 2.1.

Figure 2.1: The digraph $([1, 2, 3], \{i \to j | i < j\})$

A (directed) path in a graph is a sequence of vertices $v_1, v_2, ..., v_k, k > 0$, such that $v_i \to v_{i+1}$ is an arc for each $i, 1 \leq i < k$. In Figure 2.1, $1 \to 2 \to 3$ is a path from 1 to 3. If $v \to w$ is an arc we say $v$ is a predecessor of $w$ and $w$ is a successor of $v$.

A tree (or formally called an ordered, directed and rooted tree) is a cycle-free digraph with the following properties:

1. There is one designated node, called the root, that has no predecessors and from which there is a path to every other node.

2. Each node, except the root, has exactly one predecessor.

3. The successors of each node are ordered from left to right.

We draw trees with the root at the top and all arcs pointing downward. Thus we do not need to draw the arrows on the arcs to indicate direction and they will not be shown. We will draw the successors of each node in left-to-right order. Figure 2.2 shows an example of a tree. The nodes of this tree are labeled (or addressed) with numbers.

One can systematically label (or address) the nodes of a tree, as shown in Figure 2.2, as follows: First, we assign 0 to the root. Next, we assign 1, 2, 3, ... to the nodes immediately following the root according to the order of the edges. Then we label the remaining nodes in the following way. If $a$ is the label of a node $n$, then $a1, a2, ...$ are assigned to the nodes immediately following $n$ according to the order of the edges. We will call this labeling system as the universal address system for an ordered directed rooted tree. The universal address system provides us a way of linearly describing an ordered directed rooted tree. For example, we can order the tree shown in Figure 2.2 as follows: $0 < 1 < 1.1 < 1.2 < 1.2.1 < 1.2.2 < 2 < 2.1 < 3 < 3.1 < 3.1.1 < 3.2$. This
order is called the lexicographic order since it is similar to the way words are arranged in a dictionary.

There is a special terminology for trees that differs from the general terminology for graphs. A successor of a node is called a child, and the predecessor is called the parent. A node \( n_1 \) is called an ancestor of a node \( n_2 \) if there is a path from \( n_1 \) to \( n_2 \), and \( n_2 \) is said to be a descendant of \( n_1 \). The nodes with the same parent are called the siblings. A node with no children is called a leaf, and the other nodes are called internal nodes. For example, in Figure 2.2, the node labeled 2 is the parent of the node labeled 2.1, and the latter is the child of the former. The node labeled 3 is the ancestor of the nodes labeled 3.1, 3.2, and 3.1.1. The nodes 1, 2, and 3 are siblings. The nodes labeled 0, 1, 1.2, 2, 3, and 3.1 are internal nodes, while the other nodes are the leaves.

2.4 Regular expressions and finite automata

Let \( \Sigma \) be an alphabet and \( \Sigma' = \{ (,), \emptyset, |, * \} \) be an auxiliary alphabet, such that \( \Sigma \) and \( \Sigma' \) are disjoint. The regular expressions over the alphabet \( \Sigma \) are defined inductively as follows:

1. \( \emptyset \) is a regular expression.
2. \( \epsilon \) is a regular expression.
3. For each \( a \in \Sigma \), the string symbol \( a \) is a regular expression.
4. If \( E_1 \) and \( E_2 \) are regular expressions, then \( (E_1 E_2) \), \( (E_1 | E_2) \), and \( (E_1^*) \) are regular expressions.
5. Nothing else is a regular expression over \( \Sigma \) unless it follows from rules 1 through 4.
2.4. REGULAR EXPRESSIONS AND FINITE AUTOMATA

Let $E$ be a regular expression over alphabet $\Sigma$. Then $L(E)$ denotes the language associated with $E$, defined as follows:

1. $L(\emptyset) = \emptyset$.
2. $L(\varepsilon) = \{\varepsilon\}$.
3. For all symbols $a \in \Sigma$, $L(a) = \{a\}$.
4. For all regular expressions $E_1$ and $E_2$ over $\Sigma$, $L(E_1 E_2) = L(E_1) L(E_2)$, $L(E_1 | E_2) = L(E_1) \cup L(E_2)$, and $L(E_1^*) = (L(E_1))^*$.

To avoid cluttering in writing the regular expression, we assign priorities to the operators. From the highest to lowest precedence, we have $*$, concatenation, and $|$. Hence, one should apply $*$ first, followed by concatenation, and then $|$. For example, $ba*$ represents the language $b (a^*)$ consisting of strings that contain a single $b$ followed by zero or more $a$’s, and not the language consisting of any number of $ba$’s.

A language $L$ is regular if and only if there is a regular expression $E$ such that $L = L(E)$.

A finite automaton (FA) is a quintuple $(Q, \Sigma, \delta, I, F)$, where $Q$ is a finite set of states, $\Sigma$ is the input alphabet, $\delta : Q \times \Sigma \to 2^Q$ is the transition function, $I \subseteq Q$ is the set of initial states, and $F \subseteq Q$ is the set of final states.

The transition function $\delta$ is defined as a mapping from a state and a symbol, element of $Q \times \Sigma$, to a subset of $Q$. To describe the behavior of an FA on a string, we can extend the transition function to a function $\hat{\delta}$ that maps a state and a string, element of $Q \times \Sigma^*$, to a subset of $Q$. Namely $\hat{\delta}(q, w)$ is the state the FA will be in after reading string $w$ starting in state $q$. The formal definition is as follows:

1. $\hat{\delta}(q, \varepsilon) \to q$, and
2. for all strings $w$ and an input symbol $a$, $\hat{\delta}(q, wa) = \hat{\delta}(\hat{\delta}(q, w), a)$

For convenience, we shall use $\delta$ instead of $\hat{\delta}$ whenever the second argument is a string from here on.

A string $w$ is said to be accepted by a finite automaton $M = (Q, \Sigma, \delta, I, F)$ if $\delta(I, w) \to q'$ for some $q' \in F$. $L(M) = \{w \mid \delta(I, w) \text{ is in } F\}$ is called the language accepted by $M$. A language is regular if it is the set accepted by some finite automaton. Thus the languages accepted by finite automata are equivalent to the languages denoted by regular expressions.

Because a finite automaton is mainly used to process strings, we will also call it a string automaton. In the same way, we will also call regular languages by their other names: string languages.
The automaton is called deterministic (DFA) if and only if there is at most one initial state, and for each state \( q \in Q \) and symbol \( a \in \Sigma \) there is at most one element in \( \delta(q,a) \).

**Example 2.1 (Deterministic Finite Automaton)** Suppose we have binary alphabet \( \Sigma = \{0,1\} \), states \( Q = \{q_0,q_1\} \), initial or start states \( I = \{q_0\} \), final states \( F = \{q_1\} \), and the following transitions:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>( q_0 )</td>
<td>( q_0 )</td>
<td>( q_1 )</td>
</tr>
<tr>
<td>( q_1 )</td>
<td>( q_1 )</td>
<td>( q_0 )</td>
</tr>
</tbody>
</table>

or:

\[
\delta(q_0,0) \rightarrow q_0 \quad \delta(q_0,1) \rightarrow q_1 \\
\delta(q_1,0) \rightarrow q_1 \quad \delta(q_1,1) \rightarrow q_0
\]

This automaton, which is shown pictorially in Figure 2.3, accepts binary strings with odd parity. Strings 01, 10011 will be accepted and strings 011, 1111 will be rejected by the automaton. An example of how the above automaton processes string 10011 is as follows:

First it begins with initial state \( q_0 \). Upon reading the first symbol in the input (1), it visits state \( q_1 \). Upon reading the second symbol in the input (0), it stays at state \( q_1 \). Upon reading the third symbol in the input (0), it stays at state \( q_1 \). Upon reading the fourth symbol in the input (1), it visits state \( q_0 \). Upon reading the fifth (and the last) symbol in the input (1), it visits state \( q_1 \). The string 10011 is accepted because the last state it visits, \( q_1 \), is in the final states \( F \).

\[
\begin{array}{c}
q_0 \\
\downarrow 1 \\
q_1 \\
\uparrow 0
\end{array}
\]

Figure 2.3: An example of DFA

Let \( S \) be a positive sample set of a language \( L \) i.e. a collection of strings that are members of the language \( L \). The prefix-tree automaton (PTA) for \( S \) is a deterministic automaton that accepts precisely the set \( S \) (Angluin 1982), and is formally defined as follows: \( PT(S) = (Q, \Sigma, \delta, I, F) \) where \( Q = Pr(S) \), \( \Sigma \) is the input alphabet, \( I = \{\varepsilon\} \), \( F = S \), \( \delta(u,a) \rightarrow ua \), whenever \( u, ua \in Q \).

**Example 2.2 (Prefix-tree automaton)** Figure 2.4 shows a prefix-tree automaton (PTA) that is created from the string examples \( S : \{aaba, ab, aab\} \). A PTA is built by taking each string example to extend a path from the tree root
2.5. CONTEXT FREE GRAMMARS

to a leaf. The symbols in each string example are used to label the arcs along this path. The leaves of the PTA are marked with a double circle to indicate accepting states. A node in a PTA can be named uniquely with the path from the root to that node. In this PTA, the names are the set of all prefixes of strings in the example. To avoid cluttering the figure, integers are used as the name of the nodes. Node 1 is the starting state or root node and has empty string \( \epsilon \) as input. Clearly, this PTA will accepts only the examples.

![Figure 2.4: An example of PTA](image)

### 2.5 Context free grammars

A context-free grammar \( G \) is a quadruple \( (N, \Sigma, P, S) \), where \( N \) is the nonterminal alphabet, \( \Sigma \) is the terminal alphabet, such that \( N \cap \Sigma = \emptyset \), \( P \) is a finite set of productions, and \( S \) is a special nonterminal called the start symbol. Each production is of the form \( A \to \alpha \), where \( A \) is a nonterminal and \( \alpha \) is a string of symbols from \( (N \cup \Sigma)^* \).

If \( A \to \beta \) is a production of \( P \) and \( \alpha \) and \( \gamma \) are any strings in \( (N \cup \Sigma)^* \), we say that \( \alpha \gamma \) directly derives \( \alpha \beta \gamma \), and denote this by \( \alpha \beta \gamma \). Suppose that \( \alpha_1, \alpha_2, \ldots, \alpha_m \) are strings in \( (N \cup \Sigma)^* \), \( m \geq 1 \), and \( \alpha_1 \Rightarrow \alpha_2 \Rightarrow \alpha_3, \ldots \Rightarrow \alpha_{m-1} \Rightarrow \alpha_m \). Then we say that \( \alpha_1 \Rightarrow \alpha_m \) or \( \alpha_1 \) derives \( \alpha_m \) in \( G \).

The language generated by \( G \), denoted \( L(G) \), is \( \{ w | w \in \Sigma^* \text{ and } S \Rightarrow w \} \). A language \( L \subseteq \Sigma^* \) is a context-free language if there is a context-free grammar \( G \) with \( L = L(G) \).

If \( G \) is a context-free grammar and \( A \in (N \cup \Sigma) \), the set of derivation trees \( D_A(G) \) is recursively defined as:

\[
D_A(G) = \begin{cases} 
\{ a \} & \text{if } A = a \in \Sigma \\
\{ A(t_1, \ldots, t_k) \} & \text{if } A \to B_1 \ldots B_k \in P, \ t_i \in D_{B_i}(G), 1 \leq i \leq k \\
\{ A \} & \text{if } A \to \epsilon \in P, \ \text{where } \epsilon \text{ is the empty string} 
\end{cases}
\]
2.6 Tree automata

2.6.1 Ranked tree automata

A ranked label (or alphabet) is a label with a fixed rank (or arity). Given sets \( V_k \) of labels of rank (or arity) \( k \), \( V = \bigcup_k V_k \) is the set of all ranked labels. The set \( T_r \) of all ranked trees built with labels from \( V \) can be defined as follows:

1. A label of rank 0 (\( f/0 \) or just \( f \)) is a tree.
2. If \( f/n \) is a label of rank \( n > 0 \) and \( t_1, \ldots, t_n \) are trees, then \( f(t_1, \ldots, t_n) \) is a tree.

We represent trees by ground terms, for example the term \( a(b(a(c, c)), c) \) with \( a/2, b/1, c/0 \in V \), represents the tree shown in Figure 2.5.

```
    a
   / \
  b   c
 / \ / \n| a | c
```

Figure 2.5: A ranked tree

The terms of arity 0, 1, ..., \( p \) are respectively called constants, unary, ..., \( p \)-ary terms. The theory of tree automata emerges as a straightforward extension of the theory of string automata when strings are viewed as unary terms. For instance, the string \( aab \) over \( \Sigma = \{a, b\} \) can be viewed as a unary term \( t = a(a(b(#))) \) over the ranked labels \( V = \{a/1, b/1, #/0\} \), where \( # \) is a new constant symbol.

A deterministic ranked tree automaton or deterministic tree automaton (DTA) \( M \) is a quadruple \((V, Q, \Delta, F)\), where \( V \) is a set of ranked labels, \( Q \) is a finite set of states, \( F \subseteq Q \) is a set of final (accepting) states, and \( \Delta : \bigcup_k V_k \times Q^k \rightarrow Q \) is the transition function. For example, \( v(q_1, \ldots, q_k) \rightarrow q \), where \( v/k \in V_k \) and \( q, q_i \in Q \), represents a transition.

A DTA processes trees bottom up. A tree automaton start at the leaves and moves upward, associating along a run a state with each subtree inductively. Note that there is no initial state in a DTA. Typically when the symbol is a constant symbol \( a \), the transition is of the form \( a() \rightarrow a \). Thus the constants (leaf labels) can be considered as the “initial” states. Given a leaf labeled \( v/0 \) and a transition \( v() \rightarrow q \), the state \( q \) is assigned to it. Given a node labeled \( v/k \)
2.6. TREE AUTOMATA

with children in state $q_1, \ldots, q_k$ and a transition $v(q_1, \ldots, q_k) \rightarrow q$, the state $q$ is assigned to it. We say that a tree is accepted if a state (say $q$) is assigned to its root and this state is an accepting state, i.e., $q \in F$.

**Example 2.3 (Deterministic tree automaton)** Suppose we have the ranked labels $V = \{\text{and}/2, \text{or}/2, 0, 1\}$, states $Q = \{0, 1\}$, final states $F = \{1\}$, and the following transitions:

- $\text{and}(0,0) \rightarrow 0$; $\text{and}(0,1) \rightarrow 0$
- $\text{and}(1,0) \rightarrow 0$; $\text{and}(1,1) \rightarrow 1$
- $\text{or}(0,0) \rightarrow 0$; $\text{or}(0,1) \rightarrow 1$
- $\text{or}(1,0) \rightarrow 1$; $\text{or}(1,1) \rightarrow 1$

In the example, the labels $\text{and}/2$ and $\text{or}/2$ are the labels of the internal nodes, while the labels 0 and 1 are the leaf labels and are considered to be the "initial" states. The tree language recognized by this tree automaton is actually the set of true boolean expressions over $V$.

Intuitively, the tree automaton above works as follows: it assigns a 1 to an and-labeled node iff both of its children are 1; assigns a 0 to an or-labeled node iff both of its children are 0; assigns a 1 to an or-labeled node iff one of its children is 1; assigns a 0 to an and-labeled node iff one of its children is 0; and accepts a tree iff its root is labeled with 1. Figure 2.6 and Figure 2.7 give two examples of how this tree automaton processes trees. The tree in Figure 2.6 is accepted, while the tree in Figure 2.7 is rejected.

![Figure 2.6: A tree and an accepting run of the automaton in Example 2.3](image)

### 2.6.2 Unranked tree automata

Unranked trees have been studied since the late 60’s (Pair and Quere 1968; Takahashi 1973), see e.g. (Brüggemann-Klein, Murata, and Wood 2001) for a survey.

An unranked label is a label with a variable rank (arity). Thus the number of children is not fixed by the label. Given a set $V$ of labels in an unranked alphabet, we can define $T_v$, the set of all (unranked) trees as follows.
and

or or

\[
\begin{array}{ccc}
1 & 1 & 0 \\
0 & & 1
\end{array}
\]

Figure 2.7: A tree and a rejecting run of the automaton in Example 2.3

- $a$ is a tree where $a \in V$.
- $a(u_1, \ldots, u_n)$ is a tree, where $a \in V$ and each $u_i$ is a tree.

Given a tree $a(b, c)$, $a$ is the root label and $b$ and $c$ the children of the root. For example, an unranked tree: $a(b(c, d), b(c, e, d))$ can be visualized in Figure 2.8 (note that the two occurrences of the $b$-label have a different number of children).

![Diagram of an unranked tree](image)

Figure 2.8: An unranked tree

An unranked tree automaton (UTA) can be formalized as a quadruple $(V, Q, \Delta, F)$, where $V$ is a set of unranked labels, $Q$ is a finite set of states, $F \subseteq Q$ is a set of final (accepting) states, and $\Delta$ is a set of transitions where each transition is of the form $v(e) \rightarrow q$, where $v \in V$, $q \in Q$, and $e$ is a regular expression over $Q$. An UTA is a generalization of a tree automaton, where the alphabet $V$ is now unranked and the nodes' children in the transitions are regular expressions.

A UTA processes trees bottom up. When a leaf node is labeled $v$ and there is a transition $v(e) \rightarrow q$ such that $e$ matches the empty string, then the node is assigned state $q$. When an internal node is labeled $v_i$, its children have been assigned states $q_1, \ldots, q_n$, and there is a transition $v(e) \rightarrow q$ such that the string $q_1 \ldots q_n$ matches the regular expression $e$, then the node is assigned state $q$. A tree is accepted if the state of its root is assigned an accepting state $q \in F$.

**Example 2.4 (Unranked tree automaton)** We continue our example with the boolean expressions as in the previous example. Now we have the unranked
2.6. TREE AUTOMATA

labels $V = \{ \text{and}, \text{or}, 0, 1 \}$, states $Q = \{ 0, 1 \}$, final states $F = \{ 1 \}$, and the following transitions:

\[
\begin{align*}
\text{and}( \text{0}[1]^*0(0[1]^*)^* ) & \rightarrow 0 \\
\text{and}(1^*) & \rightarrow 1 \\
\text{or}(0^*) & \rightarrow 0 \\
\text{or}( \text{0}[1]^*1(0[1]^*)^* ) & \rightarrow 1
\end{align*}
\]

The tree language recognized by this tree automaton is actually the set of true boolean expressions over $V$, where now these labels have an arbitrary number of children.

Intuitively, the tree automaton above works as follows: it assigns a 1 to an and-labeled node iff all of its children are 1; assigns a 0 to an or-labeled node iff all of its children are 0; assigns a 1 to an or-labeled node iff at least one of its children is 1; assigns a 0 to an and-labeled node iff at least one of its children is 0; and accepts a tree iff its root is labeled with 1. Figure 2.9 and Figure 2.10 give two examples of how this tree automaton processes trees. The tree in Figure 2.9 is accepted, while the tree in Figure 2.10 is rejected.  

![Figure 2.9: A tree and an accepting run of the automaton in Example 2.4](image1)

![Figure 2.10: A tree and a rejecting run of the automaton in Example 2.4](image2)
2.7 Languages, grammars and automata

As mentioned above, a language can be characterized by defining an automaton for it. For instance, a regular language is characterized by a finite automaton (or regular expression). However, there exist another way to describe a language. That is by a grammar. Typically the generation of sentences in a language follows some fixed rules that are called a grammar. A grammar for a language will generate legal (or grammatically correct) sentences of that language. For the purpose of this thesis, it is sufficient to mention the following equivalences, in Table 2.1, between certain language classes and grammars (and automata).

<table>
<thead>
<tr>
<th>Language types</th>
<th>Grammars</th>
<th>Machine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Regular languages</td>
<td>Regular grammars</td>
<td>Finite automata</td>
</tr>
<tr>
<td>Context-free languages</td>
<td>Context-free grammars</td>
<td>Pushdown automata</td>
</tr>
<tr>
<td>Tree languages</td>
<td>Tree grammars</td>
<td>Tree automata</td>
</tr>
</tbody>
</table>

Table 2.1: Equivalence between languages, grammars, and automata

2.7.1 Relationship between formal languages

The following theorem establishes the relation between a tree grammar and a context-free grammar.

**Theorem 2.1** If $G$ is a context-free grammar then the set of derivation trees of $L(G)$ is a regular tree language. However, the vice versa is not true.

The proof of the first statement in the theorem above can be found in (Sakakibara 1992; Comon, Dauchet, Gilleron, Jacquetard, Lugiez, Tison, and Tommasi 1999). The proof of the second statement in the theorem above can be found in (Comon, Dauchet, Gilleron, Jacquetard, Lugiez, Tison, and Tommasi 1999). Below, we give an alternative proof for the second statement.

Given a tree automaton $M (V, Q, \Delta, F)$, where $V = \{a, b, c, d, e\}$, $Q = \{1, 2, 3, c, e\}$, $F = \{1\}$, and the following transitions:

- $a(2) \rightarrow 1$
- $b(3) \rightarrow 2$
- $a(c) \rightarrow 3$
- $a(4) \rightarrow 1$
- $d(5) \rightarrow 4$
- $a(e) \rightarrow 5$

The trees $t_1$ and $t_2$ in Figure 2.11 belong to $L(M)$. If these trees are derivation trees of a CFG $G (N, \Sigma, P, S)$, then $N = \{a, b, d\}$, $\Sigma = \{c, e\}$, $S = a$, and $P$ is as follows:

- $a \rightarrow b$
- $b \rightarrow a$
- $a \rightarrow c$
- $a \rightarrow d$
- $d \rightarrow a$
- $a \rightarrow e$
2.7. LANGUAGES, GRAMMARS AND AUTOMATA

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_1$: a</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>b</td>
<td></td>
</tr>
<tr>
<td>$t_2$: a</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>d</td>
<td>b</td>
</tr>
<tr>
<td>$t_3$: a</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>a</td>
<td>a</td>
</tr>
<tr>
<td></td>
<td>e</td>
<td>e</td>
</tr>
</tbody>
</table>

Figure 2.11: Tree examples

Then, $t_3$ in Figure 2.11 also belongs to CFG $G$. However, $t_3$ does not belong to regular tree language $L(M)$.

There have been previous studies that establish the relation between ranked and unranked tree languages. Barrero (Barrero 1991) argues that unranked tree languages are more natural representations than ranked tree languages for applications such as pattern recognition and computational linguistics. Barrero also studies and compares the closure properties of unranked tree languages to those of ranked tree languages.

Recently, Neven (Neven 1999) shows the correspondence between ranked and unranked tree automata. Using encoding function $\text{encode}(t)$, such as the one described in Section 4.5.3, and decoding function $\text{decode}(t)$, which is the inverse of $\text{encode}(t)$ function, for tree $t$, Neven proves the following.

**Theorem 2.2** For every unranked tree language $\tau_u$, if $\text{encode}(\tau_u)$ is recognizable then $\tau_u$ is recognizable, and for every ranked tree language $\tau_r$, if $\text{decode}(\tau_r)$ is recognizable then $\tau_r$ is recognizable.

**Recognizable tree languages** are the languages recognized by some tree automata. Neven also pointed out that the above correspondence between ranked and unranked trees does not always apply.

2.7.2 The Chomsky hierarchy

At this point, it is useful to know that language classes are commonly grouped into a hierarchy, known as the **Chomsky hierarchy** which is named after Noam Chomsky (Hopcroft and Ullman 1979). Chomsky defined four classes of languages in this hierarchy as follows.

**Type 0**: called recursively enumerable languages. A language is said to be recursively enumerable if and only if there exists an algorithm that correctly identifies whether a string belongs to the language. If a string does not belong to the language, the algorithm either runs forever, or halts
and outputs NO. Type 0 languages are Turing machines equivalent. A 
Turing machine is an abstract representation of a computing device to
give a mathematically precise definition of algorithm or “mechanical pro-
cedure”. A grammar for the type 0 language is called a phrase structure 
grammar, which is unrestricted, and its production rules are of the form:
α → β with α, β ∈ (Σ ∪ Q)*.

Type 1: called context-sensitive languages. They have productions of the 
form αAγ → αβγ with A ∈ Q, α, β, γ ∈ (Σ ∪ Q)*: A is replaced by 
β if it occurs between α and γ. The left and right neighbors of A (α and 
g) are called the context of A.

Type 2: called context-free languages. They have productions of the form 
A → β with A ∈ Q, β ∈ (Σ ∪ Q)*: The A can be replaced by β regardless 
of its context.

Type 3: called regular languages. They have productions of the form A → aB 
or A → a with A, B ∈ Q and a ∈ Σ.

Some other classes of languages also exist such as finite languages, which 
contain a finite number of strings, and recursive languages, for which there 
exists a decision procedure that tells whether or not a given string belongs to 
the language. Between these classes, the following inclusion relations hold:

finite ⊂ regular ⊂ context-free ⊂ context-sensitive ⊂ recursive ⊂ recursively 
enumerable

The above hierarchy is called the Chomsky hierarchy of string languages. 
Actually, another Chomsky hierarchy based on tree languages, which is parallel 
to the Chomsky hierarchy of string languages, exists (Comon, Dauchet, Gilleron, 
Jacquemard, Lugiez, Tison, and Tommasi 1999). It can be described as follows:

regular tree ⊂ context-free tree ⊂ recursively enumerable tree

2.8 Structured documents

2.8.1 Three types of documents

We can find some structural parts in almost all documents. For example, a book 
contains chapters, sections, paragraphs, tables, and so on, although most of it 
consists of unstructured text. In markup documents, such as Hypertext Markup 
Language (HTML) or eXtensible Markup Language (XML), the structural parts 
of a document are distinguished with markup. The markup is done by locating
2.8 STRUCTURED DOCUMENTS

COFAB INC BUYS GULFEX FOR UNDISCLOSED AMOUNT
HOUSTON, Feb 26

CoFAB Inc said it acquired Gulfex Inc, a Houston-based fabricator of custom high-pressure process vessels for the energy and petrochemical industries. CoFAB said its group of companies manufacture specialized cooling and lubricating systems for the oil and gas, petrochemical, utility, pulp and paper and marine industries.

Reuter

Figure 2.12: An example of unstructured document taken from Reuters-21578 dataset (www.research.att.com/~lewis)

the part between a start tag and an end tag. For the purpose of this thesis, similar to (Soderland 1999), we distinguish three kinds of documents:

- Unstructured document: Documents of this type contain free text or natural language text. An example of a document of this type is shown in Figure 2.12. The example shows a short message about a company acquisition. For an information extraction application, we might be interested in extracting the buyer (in this case CoFAB Inc), the company acquired (in this case Gulfex Inc), and the amount paid (in this case UNDISCLOSED). This kind of document occurs frequently in articles or news messages.

- Semi-structured document: Many documents of this type contain ungrammatical and telegraphic texts. An example of a document of this type is an ad page in the LA Times classifieds shown in Figure 2.13. Documents of this type typically contain some structures that can be recognized by its formatting. An example of a document of this type is a conference announcement taken from the DBWorld mailing list as shown in Figure 2.13. Other examples of the documents of this type are email messages, medical records, invoice letters, etc.

- Structured document: Documents of this type contain explicit delimiters that surround the relevant information. Typically this kind of document is produced from a database. An example of a document of this type is shown in Figure 2.14. The upper part is a HTML page as shown in a web browser and shows a list of people with their emails taken from the OKRA dataset. In this rendered document, the delimiters are not visible. The lower part of the figure shows the HTML file. We can see that the data
NATIONAL SCIENCE FOUNDATION WORKSHOP ON NEXT
GENERATION DATA MINING
Location: Marriott Inner Harbor, Baltimore
Date: November 1–3, 2002
Workshop Web Site: http://www.cs.umbc.edu/NGDM02/
Sponsoring Agency: US National Science Foundation

GENERAL CHAIRS:

Hillol Kargupta and Anupam Joshi
Department of Computer Science and Electrical Engineering
University of Maryland Baltimore County
hillol@cs.umbc.edu and joshi@cs.umbc.edu

1. Agoura Ready to sell! Oak Park att 3+3 hm!
Hrdwdr flrs & recessd lites. $ 439,000
(PSC:360046) 818-703-6100 or dilbeck.com

2. Bell Canyon. New Santa Fe style house.
Approx 5000 sf, 3 car garage, 5bd, 5.5ba,
library, pool, spa, view in gated community.
$ 1,545,000 For sale by owner 818-908-9992

Figure 2.13: Two examples of semi-structured document taken from the
DBWorld mailing list (www.cs.wisc.edu/dbworld) and LA Times classifieds
(www.latimes.com)
2.8 STRUCTURED DOCUMENTS

presented in this part are delimited by HTML tags. Another example of a document of this type is a XML document. Structured documents have an explicit tree structure on top of the data elements. The latter occur in (some of) the leaves. Missing data values, exceptions, and permutations in the way the data are placed can occur in structured documents. The extraction techniques developed in this thesis are aimed at structured documents.

In the literature, some other classifications have been used. Hsu and Dung (Hsu and Dung 1998) give a slightly different classification than the above one. In their classification, missing data values, exceptions, and permutations in the way the data are placed do not occur in structured documents. Instead such documents are categorized as semi-structured. This terminology is common in the database community. What we call structured is called semi-structured by that community because there is some structure but no rigid schema (see (Abiteboul 1997; Buneman 1997)).

2.8.2 HTML

HTML is the lingua franca and standard for publishing hypertext on the World Wide Web. Most readers will already be familiar with HTML. As we know, HTML uses tags such as <h1> and </h1> to structure text into headings, paragraphs, lists, hypertext links etc. Much of the data on the Web is now published in HTML pages.

When mining inside an HTML document, for instance, the structure of the document as indicated by the HTML tags will be exploited. However, the structure imposed by HTML is purely for presentation purposes. Indeed, HTML only provides tags to specify the title of the document, to partition the document into paragraphs, to indicate lists, tables, hyper-links, and so on. The HTML file of Figure 2.15, for instance, displays the page shown in Figure 2.16 and could be part of a webpage of an online bookstore where each page contains the data of some offered books. Tags are the words between brackets and determine how the text in between should be displayed. For instance, the text “List of books” between the start-tag <TITLE> and the end-tag </TITLE> is the text displayed in the title bar of the web browser. Two matching tags together with the text in between is called an element and the structures between the tags are referred to as the content. Further, <BODY> specifies the content of the HTML file and each <LI> determines a list item.

Although HTML, based on tags, is an excellent mechanism to provide platform independent browsing, it hardly imposes any semantics. Clearly Serge Abiteboul, Peter Buneman, Dan Suciu, Morgan Kaufmann and 200 are properties of the book Data on the Web and a human can infer their meaning, but
Score: 1
Name: Morten Johan Finland
First Entered: 03/20/96
Email: morten@ifi.uio.no

Score: 1
Name: Svend Finland
First Entered: 03/25/96
Email: svend.finland@internet.no

Figure 2.14: An example of structured document taken from OKRA dataset (www.isi.edu/~muslea/RISE). The upper part is the rendered view of the HTML document from a web browser, while the lower part is (a part of) the HTML document.
2.8 STRUCTURED DOCUMENTS

```html
<html>
<head>
  <title>List of books</title>
</head>
<body>
  <h3>Data on the Web</h3>
  <ul>
    <li>Serge Abiteboul, Peter Buneman, Dan Suciu</li>
    <li>Morgan Kaufmann</li>
    <li>2000</li>
  </ul>
</body>
</html>
```

Figure 2.15: An example of a HTML file.

![HTML Example](image)

Figure 2.16: A screen grab of a browser displaying the HTML file of Figure 2.15

not a computer program. Additionally, it could be possible that different books have different properties and there is no way to specify this in HTML while keeping the structure and the content of the document separated.

In our experiments, we found that many HTML documents from our datasets or from the Web contain syntactical errors, for example the omission of the closing tags or uneven nesting of tags. Most of the current web browsers tolerate these errors and they can display the HTML documents correctly as intended by the authors. Many HTML document authors are not aware or even bothered about this problem because the web browser can display the HTML documents in their intended form. Unlike the string-based methods, our tree-based methods require the HTML document to be correct syntactically because our methods work on the parse tree of the HTML documents. Thus a prepro-
cessing is needed to correct syntactical errors before the HTML documents are parsed and inputted for the learning algorithm. For this purpose, we use HTML Tidy (Raggett 1998) which is a useful program to fix HTML syntactical mistakes automatically and tidy up sloppy editing.

2.8.3 XML

Currently there is ongoing work in the area of what is called a "semantic Web" (Berners-Lee 1998). The idea here is to make the Web more understandable to computers by providing semantic tags in documents. An important impulse in this direction is given by the use of XML (eXtensible Markup Language) instead of HTML for web documents. XML (http://www.w3.org/XML) is a new standard for the specification of structured documents and data exchange on the Web developed by the World Wide Web Consortium (W3C). However, for the purpose of this thesis we can say that XML is just HTML with user definable tags. The information in Figure 2.13, for instance, could be represented in XML as shown in Figure 2.17.

```xml
<book>
  <title>Data on the Web</title>
  <author>Serge Abiteboul</author>
  <author>Peter Buneman</author>
  <author>Dan Suciu</author>
  <publisher>Morgan Kaufmann</publisher>
  <year>2000</year>
</book>
```

Figure 2.17: An XML file.

Like HTML, XML adds extra information by means of tags. Only in the latter case, this information is no longer restricted to presentation. For instance, the tag `<publisher>` indicates that Morgan Kaufmann is the publisher of the book `Data on the Web`. Hence, every application that is capable of reading XML "knows" that this bookstore offers a book with the title `Data on the Web`, authored by Serge Abiteboul, Peter Buneman, Dan Suciu, and published by Morgan Kaufmann in year 2000.

2.8.4 Document schema

As the tags in the XML document describe its semantics, XML is often called "self-describing". Nevertheless, for information extraction and integration purposes, it is convenient to have some information in advance on the structure of a collection of XML documents. Such information is provided by Document
2.8  STRUCTURED DOCUMENTS

Type Definitions (DTDs) which are essentially grammars. In brief, a DTD specifies for every element the regular expression pattern that subelement sequences of the element need to conform to. A document that conforms to a specific DTD is said to be valid w.r.t. this DTD. For the above example, such a DTD could say that each XML file consists of a sequence of books, where each book consists of a title, one or more author(s), a publisher, and a year of publication, that the order does not matter and that, for instance, no tag is compulsory. This is shown in Figure 2.18. Here CDATA means character data or string in the XML terminology. In general, the structure of a document can be quite complicated as elements can be arbitrarily nested.

```xml
<!DOCTYPE book [
  <!ELEMENT book (title, author+, publisher, year)>
  <!ELEMENT title (#CDATA)>
  <!ELEMENT author (#CDATA)>
  <!ELEMENT publisher (#CDATA)>
  <!ELEMENT year (#CDATA)>
```

A document’s DTD, hence, serves the role of a schema specifying the internal structure of the document. DTD’s are critical for realizing the promise of XML as the data representation format that enables free interchange and integration of electronic data. Indeed, without a DTD, tagged documents have little meaning. Moreover, once major software vendors and corporations agree on domain-specific standards for DTD formats, it would be possible for inter-operating applications to extract, interpret, and analyze the content of a document based on the DTD that it conforms to.

Despite their importance, DTDs are not mandatory and an XML document may not have an accompanying DTD. This may be the case, for instance, when large volumes of XML documents are automatically generated from, say, relational data, flat files, or semi-structured repositories. Therefore, it is important to build tools that infer schema information from large collections of XML documents. Garofalakis et al. (Garofalakis, Gionis, Rastogi, Seshadri, and Shim 2000), for instance, developed the tool XTRACT for inferring DTDs. However, to overcome the limited typing capabilities of DTDs, a lot of other formalisms (Lee and Chu 2000), like XML schema, XDR, SOX, Schematron, DSD, and RELAX, are currently being developed but none of them is a standard yet. Hence, a lot of work remains to be done in this area.
2.8.5 Some views of structured documents

We can view a document in several different ways. By document we mean a normal textual document which contains texts or characters and not the digitized (e.g. scanned) document which contain pixels. In one extreme, we can view a document as a sequence of characters and whitespaces. In the other extreme, we can view a document as one unique structure. Between these two extreme, in fact, we can view a document at some abstraction levels.

In (Rus and Subramanian 1997), some examples of document abstractions are presented. For instance, a document can have different syntactic topologies: bibliographic view, physical view, and logical view. In (Cohen, Hurst, and Jensen 2002), four different views of HTML document are given: token level, tree level, visual level, and geometric level views. They can be described as follows:

- **Token level view.** In this view, a document is regarded as a sequence of tokens. A token can be defined at different levels of abstraction. For example, a token can be a character, a word, a punctuation symbol, a HTML tag, a sentence, etc.

- **Tree level view.** In this view, a document is regarded as a tree. Actually the tree structure of a document can also be viewed from different levels of abstraction. For example, a tree of an HTML document can be represented as a document object model (DOM) tree that is parsed according to its definition in DTD. Another way (Yih 1997) is to construct a tree which is constructed from the DOM tree but keeps the block-defining tags and the corresponding content into a tree node. Some examples of the block-defining tags, which are assumed to create the layout blocks in the HTML document, are <H1>, <P>, and <UL>.

- **Visual level view.** This view exploits the visual characteristics of the rendered HTML file, such as font size and font type.

- **Geometric level view.** This view exploits a high-level geometric analysis of tabular information in the rendered HTML file.

Freitag (Freitag 1998b) also gives several views of documents: a term view, which is similar to the token level view above, a mark-up view, which is similar to the tree level view above, layout and typographic views, which are similar to the visual or geometric level views above, and a linguistic view, which adds the syntactic and semantic structure of the sentences in the document.

Below are some examples of how the HTML file of Figure 2.15 can be represented. Figure 2.19 gives two examples of token level views. The upper line uses a coarser token granularity than the lower line. Figure 2.20 gives two examples of tree level views. The upper figure is a DOM tree, while the lower
2.9. **Computational Learning Theory**

... `tag(<LI>) spc(_) num(2000) spc(_) tag(</LI>)` ...

... `punc(<> cap(LI) punc(>) spc(_) num(2000) spc(_) punc(<> punc(/>` ...

Figure 2.19: Two examples of a token level representation.

Figure 2.21 shows a table that contains fourteen words of different sizes. For the illustration purpose, we name the words to self-describe their own (LaTeX) font sizes. Now consider the following extraction tasks:

- Task 1: Suppose that we want to extract all words with small font size from the table. Without the font size clue, this task would be very difficult because the target fields (all word small) occur at different columns in the table. If the learner knows that the font size of the target fields is always small then this task would be easy.

- Task 2: Suppose that we want to extract the words LARGE, large, and Large from the table. If the learner is given information about the location of these words, the learner might easily learn that the target fields are actually located on one of the diagonals of the table. Without geometric information, the extraction rule is more difficult to learn because the font size and the column, where the target fields occur, are varied.

### 2.9 Computational Learning Theory

Computational learning theory (COLT) is a research field that studies the capabilities and the limitations of computer models of learning. Mitchell (Mitchell 1997) outlines some of the questions that this theory seeks to answer. These are the questions such as whether it is possible to identify classes of learning problem that are inherently easy or difficult. How many examples are needed by a learning algorithm in order to guarantee successful learning? Can the learning process be made possible or faster if the learner can ask questions or additional clues to the teacher? Can we characterize the maximum number of mistakes made by the learner before learning the target concept correctly?

There are several frameworks available for analyzing learning algorithms. In (Angluin 1996), three major currents in COLT are described:

- Inductive inference is the oldest branch of COLT. The fundamental paradigm in this area was established by Gold (Gold 1967) and is known as identification in the limit.
Figure 2.20: Two examples of a tree level representation.

<table>
<thead>
<tr>
<th>fontsize</th>
<th>small</th>
<th>tiny</th>
<th>Large</th>
</tr>
</thead>
<tbody>
<tr>
<td>normalize</td>
<td>LARGE</td>
<td>small</td>
<td></td>
</tr>
<tr>
<td>small</td>
<td>large</td>
<td>scriptsize</td>
<td>normalize</td>
</tr>
<tr>
<td>LARGE</td>
<td>huge</td>
<td>small</td>
<td></td>
</tr>
</tbody>
</table>
2.9. COMPUTATIONAL LEARNING THEORY

- Query learning, which was initiated in (Angluin 1988), models a situation where the learner may actively ask or experiment to resolve the status of particular examples.

- Probably approximately correct (PAC) learning was introduced in (Valiant 1984). The PAC paradigm put an emphasis on the statistical generation of examples and hypotheses and uses polynomial time, and in particular polynomial number of examples, as the measure of feasible learnability.

The identification in the limit model (Gold 1967). In this model, learning is viewed as an infinite process and the learner is only provided with a long (possibly infinite) sequence of positive (and negative) examples from an unknown grammar $G$. An inference algorithm $M$ is said to identify $G$ in the limit on a sequence of examples if $M$ guesses a correct grammar which is equivalent to $G$ at some point in time, and never changes its guess after this.

The query learning model (Angluin 1988). In this model, the learner is given labeled examples to learn an unknown target concept. Besides receiving labeled examples, a teacher or oracle is available to answer specific queries about the unknown grammar $G$ posed by the learner. The typical queries that are often used are as follows:

- Membership queries: does this instance belong to the target grammar $G$?

- Equivalence queries: is the hypothesis $h$ equivalent to the target grammar $G$? If not then the learner receives a counterexample $x$ such that the output of $h(x) \neq G(x)$.

The PAC learning model (Valiant 1984). Let $X$ be the set of all possible instances over which target functions may be defined, $C$ be some set of target concepts that our learner wishes to learn, and $H$ be a hypothesis class over $X$. Each target concept $c \in C$ corresponds to some subset of $X$. For example, $X$ is $\Sigma^*$, $C$ is the set of regular languages, and $H$ is the set of DFAs. In PAC learning, it is assumed that random samples from $X$ are drawn independently whose probability distribution $D$ may be arbitrary and unknown. It is required that a learning algorithm $L$ learns the target grammar:

- efficiently, which is typically measured by considering relevant parameters such as the size of examples and the size of the unknown grammar. The size of an example can be measured by the number of symbols in it, e.g., the number of nodes when examples are trees. The size of the unknown grammar can be measured by the number of states or the number of production rules or transitions.
• accurately (the error of hypothesis $h$ with respect to target concept $c$ and distribution $D$ or $\text{error}_D(h) < \epsilon$), and
• with high confidence (with probability $\geq 1 - \delta$).

The PAC learning framework can be defined more formally as follows (Mitchell 1997).

**Definition 2.1** A concept class $C$ is PAC learnable by a learner $L$ using hypothesis space $H$ if for all $c \in C$, distributions $D$ over a set of instances $X$ of length $n$, $\epsilon$ such that $0 < \epsilon < 1/2$, and $\delta$ such that $0 < \delta < 1/2$, learner $L$ will with probability at least $(1 - \delta)$ output a hypothesis $h \in H$ such that $\text{error}_D(h) \leq \epsilon$, in time polynomial in $1/\epsilon$, $1/\delta$, $n$, and $\text{size}(c)$.

The PAC learning model above defines learning directly in terms of the predictive power of the hypothesis output of the learner and concerns the computational resources needed for training. (Mitchell 1997) pointed out that a learner $L$ must learn from a polynomial number of training examples in order for $L$ to PAC-learn a concept class $C$ efficiently. In fact, a typical approach to showing that some class of target concepts $C$ is PAC learnable is first to show that each target concept in $C$ can be learned from a polynomial number of training examples and then show that there is a polynomial time algorithm that learns from these examples.

It turns out that we can bound the probability that the learner will output a hypothesis $h \in H$ such that $\text{error}_D(h) \leq \epsilon$ after a given number of training examples, even without knowing the identity of the target concept or the distribution of the training examples. Such a bound (Kearns and Vazirani 1994) can be stated as follows.

**Theorem 2.3** Let $C$ be a concept class and $H$ a hypothesis space. Let $L$ be a learning algorithm, which outputs an $h \in H$ that is consistent with input sample $S$, and runs in time polynomial in $m$ and $\text{size}(c)$, if $L$ is given $m$ labeled examples of $c$, for any $c \in C$. Then for any distributions $D$ over a set of instances $X$ of size $n$, and any target concept $c \in C$, if $L$ is given as input a random sample of $m$ examples, where $m$ satisfies

$$m \geq (1/\epsilon)(\ln|H| + \ln(1/\delta))$$

then $L$ is guaranteed to find a hypothesis $h \in H$ that with probability at least $1 - \delta$ obeys $\text{error}_D(h) \leq \epsilon$.

Note that Theorem 2.3 does not necessarily claim that $L$ is an efficient PAC learning algorithm. In order to assert that $L$ is an efficient PAC learning algorithm, $m$ also has to be polynomially bounded in $n$, $\text{size}(c)$, $1/\epsilon$ and $1/\delta$. 
2.10 Summary

In this chapter, we have described some concepts and definitions that can serve as background for the thesis. We have elaborated the description of tree automata and have discussed several types of documents. This thesis focuses on the application of tree automata inference based methods for extracting information from structured documents. In this thesis, our tree-based methods work with samples represented in the document object model (DOM).

In addition, we have described the relation between tree languages and context-free languages and the correspondence between ranked and unranked tree automata. Finally, we have also described three major currents from the computational learning theory field. We will use two of these major currents, the identification in the limit model and the PAC model, to analyze our learning algorithms.

In the next chapter we will describe the core topics of this thesis: information extraction and tree automata inference methods.
Chapter 3

Information extraction and grammatical inference

3.1 Introduction

In this chapter, our main purpose is two-fold: to give an overview of information extraction (IE) and of the current state-of-the-art in tree automata inference methods.

In the first part of the chapter, we start with a definition of IE and its relation to some related technologies, and explain that IE can be viewed as a web mining application. Next we give some possible applications of IE, including the recent disagreement regarding the importance of extracting information from HTML pages. After that we propose a taxonomy of IE to better situate our work. We actually categorize IE into three major categories: classical IE, IE from semi-structured data, and IE from structured data. Then we give an overview of each category. Here, our intention is just to give a general overview and a summary of the current research giving an emphasis on IE from structured data. A more detailed review of structured IE methods is described in Chapter 6 about related work. We end the first part of the chapter with a discussion.

In the second part of the chapter, we first give an introduction on grammatical inference and describe some possible applications. Next, we describe a setting of learning from positive examples only. Actually this is the setting that we use for our tree automata inference methods. We then continue with describing the inference of regular grammars and some algorithms to induce them. Due to the size of this field, here we focus on the regular grammar inference methods that are related to our tree automata methods. Actually our methods are basically extending these string-based methods. The next section
gives an overview of the current state-of-the-art of the tree automata inference methods that are based on ranked trees. We show that these tree automata inference methods are basically an upgrade from the existing string-based automata inference methods. We also give a characterization of these tree-based methods.

Finally, we conclude the chapter with a summary.

3.2 Information extraction (IE)

3.2.1 Definition

Information extraction (IE) is the task of extracting a portion of text or data from larger documents and can be defined informally as follows:

Definition 3.1 (Information extraction)

Given: A set of documents, possibly from the same domain or topic,

Task: Extract the fragments of interest.

As noted previously in Chapter 1, there is a complex interaction between data mining and information retrieval / extraction. Below we discuss this issue in more detail.

According to (Cowie and Lehner 1996), IE has the goal of transforming a collection of documents, usually with the help of an information retrieval (IR) system, into information that is more readily digested and analyzed. IE aims to extract relevant facts from the documents while IR aims to select relevant documents (Pazienza 1997). While IE is interested in the structure or representation of a document, IR typically views the text in a document just as a bag of unordered words (Wilks 1997). Thus, in general IE works at a finer granularity level than IR does on the documents. However, the difference between the two becomes blurred if the interest of IR is in extraction (Pazienza 1999), and when used in the context of vague forms of information in which a full text IR system can provide some IE features (Wilks 1997).

Building IE systems manually is not feasible and scalable for such a dynamic and diverse medium as the Web (Mustea, Minton, and Knoblock 1998). Due to this nature of the Web, most IE systems focus on a small number of web sites to extract. Others use machine learning or data mining techniques to learn the extraction patterns or rules for web documents semi-automatically (Kushmerick 1999). Within this view, web mining is part of the (web) IE process. Other views regarding the relationship between (web) IE and Web mining also exist. The results of the IE process could be in the form of a structured database or could be a compression or summary of the original text.
or documents. In the former case, IE can be viewed as a kind of pre-processing stage in the web mining process, namely the step after the IR process and before the data mining techniques are applied. In a similar view, IE can also be used to improve the indexing process, which is part of the IR process. Conversely, in the latter case, one can argue that IE is an instance of text or web mining since the summary or the compressed form of a document is a new form of information that does not exist before. However, we adhere to the view that web mining is used to improve web IE (web mining is part of IE).

### 3.2.2 Applications of IE

There are many applications of information extraction. At the most basic level, IE functions are extractors and the extracted data becomes the end product. Some examples are price extractors and named entity extractors, which could be directly used by the price comparison agents (Doorenbos, Etzioni, and Weld 1997), keyword and keyphrase extractors (Turney 1997), where the extracted keyphrases could be directly useful for creating keywords for scientific articles or for summarizing a document (Nanba and Okumura 1999). In database applications, IE systems are often used to transform unstructured or semi-structured texts into structured texts. This is either used to transform HTML into XML (Lin, Fu, and Han 2000) or to use the extracted information to fill a database. In these examples, the IR system is used as the front-end and as a part of a larger IE system.

Since information extraction can be viewed as an enabling technology, we could also include IE system as a front-end or pre-processor in larger systems. Actually the keyphrase extractor above could also be used for the indexing process in information retrieval systems. Some other examples include using IE system as a front-end for data/information integration systems (Levy, Knoblock, Minton, and Cohen 1998), and as a front-end for data mining systems for text and the Web (NaM and Mooney 2000).

Encouraged by these successful applications, we believe that IE can help to improve information access.

Recently a controversy emerged regarding the claim that the adoption of the XML standard will make information extraction from HTML documents significantly less important. First, note that IE problems for other less-structured and unstructured domains still remain. There are also many counter arguments against this view. Some of these are as follows:

- Knoblock and Minton (Levy, Knoblock, Minton, and Cohen 1998) argue that the need for HTML wrappers will remain strong because (i) there will always be exceptions where the information providers only share their data selectively, (ii) possible disagreement on the granularity of the modeled information, and (iii) the extended use of legacy pages.
It is argued in (Cohen and Fan 1999; Kushmerick 2000b) that currently only relatively little data are exported to XML and that it remains to be seen if the huge amount of non-XML legacy data can be converted to XML. Another problem with XML is that XML forces information providers and users to agree on the ontology.

Considering that XML documents are based on varying DTDs or XML schemas, the current problem can only be reduced but not completely solved (Kuhlins and Tredwell 2002).

XML has been so far of little help to overcome some of the limitations of HTML (Crescenzi, Mecca, and Merlaklo 2001). Thus extracting data from HTML remains a relevant task.

Although XML becomes more common to be used as internal data representation, HTML pages will still exist because outsiders typically received data in the XSL/CSS-script generated HTML pages. Thus, wrapping HTML pages is expected to remain an important topic (May and Lautsen 2000).

Indeed, the emergence of XML as data exchange standard will make the IE problem from XML documents easier but will certainly not make IE from HTML documents significantly less important. When the schema and a query language are available, the problem of extracting data from XML documents can be solved by writing a query for the schema. However, we still need to extract data from HTML documents.

### 3.2.3 Taxonomy of IE

Building an extractor for a set of web pages is an ad-hoc process and the resulting system is typically only applicable to a specific set of web pages. Currently, there is no single system that is superior for all types of documents. This is because documents exist in various kinds, ranging from rigidly structured ones, such as databases, to completely unstructured ones, such as free texts. Moreover, each type of document often requires the application of different kinds of technologies to extract.

For example, in unstructured text, it is very useful if the IE system “knows” that the text that it should extract is of type noun. In this case, linguistic knowledge or pre-processing tools such as a part-of-speech tagger can be very helpful. For semi-structured texts, the linguistic pre-processing tools that are useful for free texts are rendered almost useless. This is because, as mentioned previously, semi-structured texts contain ungrammatical and telegraphic texts. Typically IE systems that work with this kind of documents analyze and use the surrounding, that is the left and the right neighbors, of the fields of interests.
3.2. INFORMATION EXTRACTION (IE)

If the fields of interest always have the same neighbors then these neighbors could be used to locate the fields of interest in other (similar) documents. For structured texts the same strategy, that is, analyzing the surrounding to locate the fields of interests, can be applied. In structured texts, however, other strategies could also be applied, such as using the underlying structure of how the texts are arranged in a document.

Thus it is useful to categorize the IE problem into three distinct types according to the type of the document: IE from unstructured texts (often called classical or traditional IE), IE from semi-structured texts, and IE from structured texts. Our proposed taxonomy for the IE problem can be seen in Figure 3.1.

There are many alternatives regarding the taxonomy of IE. To the best of our knowledge, the most detailed taxonomy of IE so far is proposed by Crespo, et al. (Crespo, Jamnik, Neuhold, Rys, and Studer 2002). However for the purpose of this thesis, we situate our research using the taxonomy in Figure 3.1.

Within each type of document, we categorize the IE problem further based on how the extraction system was built. Some IE systems may be built manually using a knowledge engineering approach. Other IE systems are built (semi-)automatically, typically using machine learning techniques or heuristics. Because this thesis focuses on structured documents, we further divide the taxonomy below the structured document category to better situate our work.

There are two types of work in the manual approach: the pure knowledge engineering approach and the query language approach. An advantage of the query language approach over the pure knowledge engineering approach is that the former requires less knowledge to build. The semi-automatic approach is mainly based on the wrapper induction approach, which typically uses machine learning techniques. Finally the automatic approach uses either unsupervised learning techniques or a heuristics approach.

The next subsections present a general overview of the characteristics of each document category.

3.2.4 Classical or traditional IE

As mentioned above, classical or traditional IE tasks from unstructured natural language texts typically use various forms of linguistic pre-processing. An example domain investigated in the Sixth Message Understanding Conference (MUC-6) (MUC-6 1995), is “Management Succession”. Given an article, the tasks are to extract the name of the new company officers, the old officers, the company name, and the title of the succeeded position.

Classical or traditional IE research, with roots in the NLP community, has been studied for quite a long time (Cowie and Lehnert 1996; Wilks 1997). We
Focus of this work

Information extraction (IE)

IE from unstructured text (MUC tasks)
Rely on HTML/XML tags. The information is typically arranged in a kind of tabular format

IE from semistructured text (Web data)
Rely on non-linguistic structures. Typically a mix between HTML tags and ungrammatical texts

IE from structured text (Web data)
Use various forms of linguistic preprocessing

Knowledge engineering and query languages
Manual (Semi-)
automatic

Wrapper induction or by learning
Unsupervised and heuristic

Focus of this work

Manual
(Semi-)

Automatic

Manual

(Semi-)

Automatic

Manual

(Semi-)

Automatic

Manual

(Semi-)

Automatic

Manual
could say that the Advanced Research Projects Agency (ARPA) helped creating the field (classical IE) because the evaluations of IE cannot be separated from the ARPA sponsored Message Understanding Conferences (MUCs) and the TIPSTER IE project (Wills 1997; Appelt and Israel 1999). MUCs and TIPSTER are competitive environments that seek to improve IE and IR technologies (Cowie and Lehner 1996; Cardie 1997). Classical IE usually relies on linguistic pre-processing such as syntactic analysis, semantic analysis, and discourse analysis (Soderland 1999; Muslea 1999; Kushmerick 1999). Indeed, classical IE could be called a core language technology (Wills 1997).

For more detailed information about classical IE and the issues of IE from unstructured text we refer to the survey articles of (Cardie 1997; Cowie and Lehner 1996; Appelt and Israel 1999) and the conference proceedings of (Paziienza 1997; Paziienza 1999).

### 3.2.5 IE from semi-structured data

With the increasing popularity of the Web as a medium for disseminating information and the work on intelligent information agents (Green, Hurst, Nangle, Cunningham, Somers, and Evans 1997) there is a need for structural IE systems that extract information from (semi-) structured documents. Indeed, one important task that has to be performed by intelligent information agents is extracting data from several places in the Internet. Intelligent agents need this data and information in order to make decisions autonomously. Thus wrappers are an important part of intelligent agents.

Structural IE research is different from the classical one as it usually utilizes the meta-information (e.g. HTML tags (Soderland 1999), simple syntaxes (Kushmerick 1999), or delimiters (Muslea 1999)) that are available inside the (semi-) structured data. Structural IE approaches that use machine learning and do not use linguistic constraints are termed *wrapper induction* (Kushmerick, Weld, and Dootenbos 1997) and structural IE systems are often called wrappers.

Some structural IE systems are built manually by a knowledge engineering approach, examples are (Huck, Funkhauser, Aberer, and Neuhodl 1998; Hamner, García-Molina, Cho, Crespo, and Aranha 1997). Other manual approaches are built using ontologies (Embley, Campbell, Liddle, and Smith 1998), where domain experts create a specific ontology for the problem domain before the system can be used. A major characteristic of the manual IE systems is that the usage requires some non-trivial level of expertise, either in the form of ontology writing or extraction rules specification.

To address these difficulties there has been increasing interest in applying machine learning (ML) techniques to information extraction. More and more structural IE systems for the Web are built (semi-)automatically using machine learning techniques or other algorithms. Wrapper induction has already been
addressed by several authors; some examples are (Hsu and Dung 1998; Musleà, Minton, and Knoblock 2001; Kushmerick 2000a; Freitag and Kushmerick 2000). These IE systems are built by using machine learning or data mining techniques that learn extraction rules from the annotated corpora. Several machine learning techniques have been used e.g. inductive logic programming (Junker, Sintek, and Rinck 1999; Freitag 1998a; Califf and Mooney 1999), propositional rule learning (Musleà, Minton, and Knoblock 2001; Soderland 1999; Kushmerick 2000a), naive-Bayes (Frank, Pynter, Witten, Gutwin, and Nevill-Manning 1999), Hidden Markov Models (Freitag and McCallum 1999), multi-strategy approaches (Freitag 2000), and boosting (Freitag and Kushmerick 2000). Other techniques include finite state transducer induction (Hsu and Dung 1998), and finite automata induction (Chiklovskii 2000; Chiklovskii, Ragedi, and de Rijke 2000).

For more detailed information about the IE work on semi-structured texts, we refer to recent surveys such as (Musleà 1999; Soderland 1999; Kushmerick 2000a).

3.2.6 IE from structured data

Much research on IE from structured data originates from the database community and one of the driving forces is the work on information integration (Florescu, Levy, and Mendelzon 1998; Levy, Knoblock, Minton, and Cohen 1998; Wiederhold 1996). These information integration systems collect and integrate data and information from several web sites so that the users can query the unified view of the integrated data. This is very useful for the users since many web sites contain partial or overlapping information. By integrating these information taken from several web sites, the users can get more complete and non-redundant information about a specific topic. In order to extract the data from these web sites, wrappers are used. Typically a wrapper is needed for each web site because the data on the Web is present in different kinds of forms and structures. Because of the nature of the data to be extracted, IE from structured data is often called data extraction to differentiate it from information extraction.

Some structured IE systems may be built manually using a knowledge engineering approach (Chawathe, Garcia-Molina, Hammer, Ireland, Papakonstantinou, Ulman, and Widom 1994; Atzeni and Mecca 1997). In this approach, the developers manually craft the extraction rules that could be used for the actual extraction of the fields of interest. This approach typically requires the users to write the extraction rules themselves in a kind of pattern languages, which require specific technical skills. Another manual approach is by means of systems that use templates that specify the document structure and/or information to be extracted (Hsu and Yih 1997). This requires some non-trivial level of expertise for template specification.
3.2. INFORMATION EXTRACTION (IE)

Throughout the manually built structured IE systems, there exists several query languages supporting the extraction of information from structured data (Bry and Schaffert 2002; Abiteboul, Quass, McHugh, Widom, and Wiener 1997; Baumgärtner, Flesca, and Gottlob 2001). The benefit of query languages over manual systems that are built with knowledge engineering techniques are as follows. Firstly, the users do not need special skill to write extraction rules, for example in a pattern language based on regular expressions. Secondly, the users do not need to see the syntax of the structured documents. These two benefits are made possible with a fancy graphical user interface (GUI) that is provided by query languages. However, to get the benefits from their capabilities, their use requires at least the skill to use the GUI.

However, building IE systems manually is a tedious task, error-prone, time consuming, and requires a special skill. Thus it is not feasible and scalable for such a dynamic and diverse medium as the Web (Muslea, Minton, and Knoblock 1998; Kushmerick 1999). As argued for example in (Muslea, Minton, and Knoblock 2001; Kushmerick 2000a), there is a need for systems that can learn to extract information from a few annotated examples. IE systems that are built (semi-)automatically often use machine learning or related techniques. Typically these systems only require the users to label the fields of interest as examples to the system. Compared to the manual approaches described above, this approach requires less user effort and knowledge.

Besides the above approaches, there are IE systems that are built in a completely automatic way. These systems are typically based on some assumptions about the structured documents. Some available systems use heuristics (Crescenzi, Mecca, and Merialdo 2001), others use unsupervised learning approaches (Leman, Knoblock, and Minton 2001). Usually the systems that are built in this way are only able to extract document blocks, for example, a paragraph or a record. Thus these systems extract all fields in a document block instead of extracting a specific field of interest.

A more detailed description of IE from structured data is given in Chapter 6.

3.2.7 Discussion on IE systems

Usually systems that work on semi-structured data are the most flexible ones. The reason is that semi-structured data lies in between the two document extremes: structured and unstructured. Thus it has characteristics that are a mix of those of structure and unstructured data. As a consequence, methods that work on this kind of data have to be flexible enough to deal with characteristics of both kinds of data. This is the reason why systems that work on the semi-structured data are also able to work on both structured and unstructured data. However their performance on structured and unstructured data might not be as good as the performance of IE systems that are made specially for these data types.
The other way around also holds. Typically IE systems that are specifically made for either structured or unstructured data are the least flexible ones. One reason for this is that they need some (document) type-dependent features that are exclusive to a specific document type. For example, IE systems that are designed to work with unstructured texts typically process the free texts using the linguistic processing tools and needs several pre-processing steps, such as tokenization, morphological and lexical processing, syntactic analysis, and domain analysis (Appelt and Israel 1999). These kind of IE systems will not work for ungrammatical texts. Another example is for IE systems that are designed to work with structured data. Some of these systems require structural information, such as the tree structure of the document, in order to work. Thus if the texts contain no structural information, then these systems simply cannot work.

BWI (Freitag and Kushmerick 2000), a structural IE system that does not use any linguistic knowledge, has been applied to do IE from the whole spectrum of structured to unstructured data domains and has shown to be competitive to other IE methods. This development might suggest that linguistic knowledge is not very useful for IE and that one semi-structured representation is sufficient for all document types. However, there is some evidence against this view. The first evidence is that some recent work such as (Káthak, Smarr, and Elkan 2002; Ciravegna 2001; Eliassi-Rad and Shavlik 2001) suggest that even in ungrammatical semi-structured texts, linguistic knowledge is useful. The second reason is, as will be argued in Chapter 6, that using a specialized representation can achieve better performance than when using only one representation for all types of documents.

We will discuss some recent development in IE research, with an emphasis on structured IE systems, in more detail in Chapter 6.

### 3.3 Grammatical inference

#### 3.3.1 Introduction

\[
S \rightarrow 0S1 \\
S \rightarrow \epsilon
\]

Figure 3.2: An example of context-free grammar that produce \( L = 0^n1^n \)

As mentioned previously, a grammar is a representation of the syntax of a language. Given a grammar for a language, we could use it to generate legal (or syntactically correct) sentences that belong to the language. For example, the (context-free) grammar of Figure 3.2 generates strings of 0s followed by the
3.3. GRAMMATICAL INFERENCE

same number of 1s: 01, 0011, ..., 0^n. Grammatical inference is the reverse process of the sentence generation above, and can be stated as follows.

Definition 3.2 (Grammatical inference problem)

Given:
- some sentences and for each of them, whether it belongs to the language.
- a criterion for successful inference.

Task: infer a grammar or its equivalent representation (e.g. an automaton) that generates the language.

This definition is a slightly simplified version of the inductive inference problem specification proposed in (Angluin and Smith 1983). In our example, the process, when learning from positive examples only, is as follows. Given example strings of the form 01, 0011, ..., 0^n, the task is to infer a grammar that generates them (that is: S → 0S1 and S → ε).

In other words, grammatical inference refers to the process of learning rules from a set of labeled examples. It is a part of machine learning and belongs to a class of inductive inference problems (Angluin and Smith 1983) in which the target domain is a formal language (a set of strings over some alphabet Σ) and the hypothesis space is a family of grammars. In the literature, it is also often referred to as automata induction, grammar induction, or automatic language acquisition. It is a well-established research field in AI that goes back to Gold's work (Gold 1967).

The grammatical inference process aims at finding a grammar or an equivalent representation that is compatible with the examples. The compatibility with the examples depends on the applied quality criterion. Quality criteria that are generally used in the grammatical inference work are:

- identification in the limit (Gold 1967),
- probably approximately correct (PAC) learning (Valiant 1984).

3.3.2 Applications of grammatical inference

Grammatical inference has many applications. One of the classical application of grammatical inference, which is due to the research by Gold (Gold 1967), is to study and construct a formal model of human language acquisition. Another classical application field is syntactic pattern recognition (Gonzalez and Thompson 1978; Fu 1982).

More recently some other applications emerged. DFA have been proposed for opponent modeling (in a game-theoretic sense) (Peterson and Cook 1998).
In (Ahonen 1996), DFA have been used to learn the content model of SGML documents. DFA have also been used by a robot trying to model its environment (Rivest and Schapire 1993).

Probabilistic string-based methods such as Hidden Markov Models have been used extensively in the speech processing and natural language processing communities (Rabiner 1989; Jelinek 1998). Probabilistic string-based methods have also been applied to recognizing patterns in biological sequences such as DNA and proteins (Baldi and Brunak 1998).

Probabilistic context-free grammars have been used for modelling RNA (Sakakibara, Brown, Hughey, Miyazawa, Sjölander, Underwood, and Haussler 1994). probabilistic tree-based methods have been applied to predicting protein structure (Abe and Mamitsuka 1997), to tackle ambiguity in natural language parsing, document analysis and modeling, etc. (Charniak 1993).

Since grammatical inference is a subfield of machine learning, many knowledge acquisition problems that require automatic processing, such as data mining, etc., can be regarded as potential applications of grammatical inference methods. Thus, there are many potential problems where grammatical inference methods can be used.

### 3.3.3 Learning from positive examples

In information extraction, the learner is typically only presented with positive examples $S^+$. One reason is that providing negative examples can be tedious and impractical. For example, when interested in the extraction of a name field in a set of documents, a typical approach is to provide positive examples where one of the name fields is labeled as the field to be extracted. As the document can have several name fields, the other name fields (that are not marked) cannot be considered as negative examples. Actually a “closed world” solution for this is possible: the user is required to label the field of interest completely, so that the other fields (other than the marked ones) can be assumed to be negative examples. However, in this thesis we choose to learn from positive examples only.

It is well-known that learning from positive examples only is more difficult than learning from a complete presentation of the examples. A complete presentation of an unknown grammar $G$ is an infinite sequence of ordered pairs $(w, l)$ from $\Sigma^* \times \{0, 1\}$ such that $l = 1$ if and only if $w$ is generated by $G$ and each $w$ appears at least once in the sequence. Gold (Gold 1967) has shown that no member of the superfinite class of languages can be identified in the limit from positive presentation. A language in the superfinite class of languages is a language that contains all the finite languages and at least one infinite language. The problem is overgeneralization: any automaton for a superset of the target language will also accept the target language. If during the inference process such a superset is hypothesized, using the rest of the positive examples will not
be able to correct the error. The class of regular languages and context-free languages are superfinite. Thus the class of regular languages and context-free languages cannot be identified in the limit from positive presentation. A table that summarizes Gold’s result (Gold 1967) can be seen in Table 3.1.

<table>
<thead>
<tr>
<th>Class of languages</th>
<th>Learnable from positive and negative examples?</th>
<th>Learnable from positive examples?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Recursively enumerable</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Context sensitive</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Context free</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Regular</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Superfinite</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Finite cardinality</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

Table 3.1: Gold’s result of the learnability of language classes

Some obvious facts that can be derived from Table 3.1 are: If a class of language is identifiable from the limit with respect to a language learnability model (using negative and positive or only positive examples), then the same holds for any subclass; if a class is not identifiable in the limit, then the same holds for any superclass.

However, this does not mean that we cannot do reasonable information extraction by providing positive examples only. Indeed, there are several non-trivial classes of languages that can be learned from positive examples only (Angluin 1980). Moreover, there are several ways to compensate for the lack of negative examples (Knuth 1996). Either, we can restrict the language classes that we want to learn or we can provide more information, such as the structure or the probability, to the method.

### 3.3.4 Inference of regular grammars

There is a large body of work on grammatical inference, for recent excellent surveys see e.g., (Murphy 1996; Sakakibara 1997; Parekh and Honavar 1998). In this section, we will only review regular grammar inference methods that are closely related to the tree automaton inference methods that are the focus of this thesis. Our tree automata based methods and some of the tree-based methods that are described in the next section are actually extensions of these string-based methods. The intention is to provide a background for the tree automata inference methods that will be discussed in Chapter 4.

In regular grammar inference, we have a finite alphabet $\Sigma$ and a regular language $L \subseteq \Sigma^*$. The problem can be stated as follows.

**Definition 3.3 (Regular grammars inference)**
Given: a set of examples that are in the language \( S^+ \) and a (possibly empty) set of examples not in the language \( S^- \).

Task: infer a deterministic finite automaton (DFA) \( A \) that accepts the examples in \( S^+ \) and rejects the examples in \( S^- \).

In (Anghin 1980), Anghin gives the characteristics of the classes of language that can be learned from positive examples only. As mentioned previously, the problem when learning from positive examples only is overgeneralization. One approach for avoiding overgeneralization of the hypothesis during the inference process is to define a characteristic sample for the language. A characteristic sample of an automaton \( A \) is a finite sample \( S \subseteq L(A) \) such that \( L(A) \) is the “smallest” language that contains \( S \).

Based on Anghin’s result, several non-trivial subclasses of regular languages have been invented. Three important subclasses of regular language that are learnable from positive examples only are: \( k \)-reversible languages (Anghin 1982), \( k \)-contextual languages (Muggleton 1990), and \( k \)-testable languages (García and Vidal 1990). These three classes of languages have similar characteristics: the strings of the language can be constructed by concatenating strings of length \( k \), for \( k = 0, 1, 2, \ldots \). Below a language characterization for the above classes is given.

### 3.3.4.1 Inference of \( k \)-reversible languages

**Definition 3.4** A regular language \( L \) is \( k \)- reversible if and only if for all strings \( u_1, u_2, v \) and \( v \), if \( u_1v \) and \( u_2v \) are in \( L \) and \( |v| = k \), then \( T_L(u_1v) = T_L(u_2v) \).

The intuition of the \( k \)-reversible languages is as follows. If two strings in \( L \) end with a common suffix \( v w \) of length at least \( k \), then the state reached by a DFA for \( L \) after reading the \( k \) common symbols is the same, i.e., \( \delta(I, u_1v) = \delta(I, u_2v) \).

The \( k \)-RI algorithm (Anghin 1982) is able to infer a \( k \)-reversible automaton from positive examples only. Given an input \( S \) of positive strings examples, the \( k \)-RI algorithm first constructs the prefix-tree automaton (see Section 2.4) for \( S \), whose states correspond to the trivial partition. The trivial partition of a set \( S \) is the class of all singleton sets \( \{s\} \) such that \( s \in S \). Then, it merges any two states \( q_1 \) and \( q_2 \) if one of the following conditions for \( k \)-reversibility holds (Muggleton 1990):

1. There exist two transitions \( \delta(q_3, v) \rightarrow q_1 \) and \( \delta(q_5, v) \rightarrow q_2 \), where \( v \in \Sigma \) is a symbol.
2. There exists a string \( s \) of length \( k \) and states \( q_a \) and \( q_b \) such that \( \delta(q_a, s) \rightarrow q_1 \) and \( \delta(q_b, s) \rightarrow q_2 \), and either
3.3. GRAMMATICAL INFERENCE

(a) \( q_1, q_2 \in F \), or
(b) \( \delta(q_1, v) = \delta(q_2, v) \), where \( v \in \Sigma \) is a symbol.

When no such pair of state blocks exists, \( k \)-RI then constructs and outputs an automaton that is formed by the current (final) partition.

3.3.4.2 Inference of \( k \)-contextual languages

**Definition 3.5** A regular language \( L \) is \( k \)-contextual if and only if for all strings \( u_1, u_2, w_1, w_2 \) and \( v \), if \( u_1v w_1 \) and \( u_2v w_2 \) are in \( L \) and \( |v| = k \), then \( T_L(u_1v) = T_L(u_2v) \).

The intuition of the \( k \)-contextual languages is that the existence of two equal sequences (of length \( k \)), implies that the resulting state should be the same. Thus the \( k \)-contextual languages have a weaker condition for state merging than the \( k \)-reversible languages. This enables the inference of a \( k \)-contextual automaton from a single string example only. If we compare the definition of the \( k \)-contextual languages to that of the \( k \)-reversible languages above, it is quite clear that any \( k \)-contextual language is \( k \)-reversible (Muggleton 1990).

The \( KC \) algorithm (Muggleton 1990) is able to infer in the limit a \( k \)-contextual automaton from positive examples only. The way this algorithm works is similar to the \( k \)-RI algorithm above. Given an input \( S \) of positive string examples, the \( KC \) algorithm constructs the prefix tree automaton for \( S \). Then, it merges any two states \( p_k \) and \( q_k \) if the following condition for \( k \)-contextuality holds: if there exist two states \( p_0, q_0 \) and any string \( v \) with \( |v| = k \) such that \( \delta(p_0, v) = p_k \) and \( \delta(q_0, v) = q_k \), then \( p_k \) and \( q_k \) are merged.

When no such pair of state blocks exists, \( KC \) then constructs and outputs an automaton that is formed by the current (final) partition.

Below we formalize some characteristics of the \( KC \) algorithm: it is efficient and incremental in the sense that the algorithm produces a gradually changing output given progressive augmentation of positive presentation of the language \( L \). A *positive presentation* of \( L \) is an infinite sequence \( \sigma = w_1, w_2, w_3, \ldots \) of strings such that the set \( \{w_1, w_2, w_3, \ldots \} \) is precisely \( L \).

**Theorem 3.1** (Muggleton, 1990) The algorithm \( KC \) can be implemented to run in time \( O(n) \) where \( n \) is one more than the sum of the lengths of the input strings.

**Theorem 3.2** (Muggleton, 1990) Given a fixed natural number \( k \), the algorithm \( KC \) is incremental on input \( k \) and any positive presentation of some \( k \)-contextual language \( L \).

Let \( S \) be a set of strings over \( \Sigma \), where \# \( \notin \Sigma \). The set of \( k \)-grams of \( S \) is defined as: \( \text{grams}(S, k) = \{u \mid u \text{ is a substring of } \#^{k-1}s\#, |u| = k, s \in S \} \).
Informally, a $k$-gram is defined as a string of length $k$. Let $G$ be a set of $k$-grams over alphabet $\Sigma$. The language generated by $G$ is defined as: $L(G) = \{w \mid \text{grams}(\{w\}, k) \subseteq G\}$.

A variant of the $k$-contextual algorithm that merges $k$-grams can be found in (Ahoen 1996). The following two theorems tell that a set of $k+1$-grams is sufficient to represent a $k$-contextual language.

**Theorem 3.3 (Ahoen, 1996)** If a language $L$ is $k$-contextual, then there exists a set $G$ of $k+1$-grams such that $L(G) = L$.

**Theorem 3.4 (Ahoen, 1996)** Let $G$ be a set of $k+1$-grams. Then $L(G)$ is $k$-contextual.

### 3.3.4.3 Inference of $k$-testable languages

Garcia and Vidal (Garcia and Vidal 1990) define a $k$-testable language in the strict sense ($k$-TLSS) by the regular expression:

$$L = (I \Sigma^*) \cap (\Sigma^* F) - (\Sigma^* T \Sigma^*)$$

where $\Sigma$ is the alphabet, $I, F \subseteq \cup_{i=1}^{k-1} \Sigma^i$ are the sets of initial and final substrings respectively, and $T \subseteq \Sigma^k$ is a set of forbidden substrings. Thus the strings in the $k$-testable languages can be characterized as follows: they start with substrings in $I$ and end with substrings in $F$, and they do not contain any substrings of length $k$ which is in $T$.

The class of $k$-testable languages are actually equal to the class of $k$-contextual languages. $k$-testable languages define the languages with the help of forbidden substrings while $k$-contextual languages use the allowed substrings (Ahoen 1996).

The $k$-TSSI algorithm (Garcia and Vidal 1990) takes as input a set of strings that are positive examples. From this input strings, it finds the sets of initial $I$, final $F$, and forbidden substrings $T$. The output automaton $(Q, \Sigma, \delta, q_0, q_f)$ is constructed as follows. The final substrings $F$ directly become the accepting states $q_f$, while the states $Q$ and transitions $\delta$ are collected from:

- for $m = 1$ to $k - 1$
  - for all $a_1 \ldots a_m \in I$
    - $Q = Q \cup \{a_1 \ldots a_m\}$
    - $\Delta = \Delta \cup \{\delta(a_1 \ldots a_{m-1}, a_m) \rightarrow a_1 \ldots a_m\}$
- for all $a_1 \ldots a_k \in (\Sigma^k - T)$:
  - $Q = Q \cup \{a_2 \ldots a_k\}$
3.3 GRAMMATICAL INFERENCE

- \( \Delta = \Delta \cup \{ \delta(a_1 \ldots a_{k-1}, a_k) \rightarrow a_2 \ldots a_k \} \)

Garcia and Vidal (Garcia and Vidal 1990) prove that \( k \)-TSSI algorithm identifies any \( k \)-TLSS in the limit from positive data. While \( k \)-testable languages are less expressive than \( k \)-reversible languages, learning the automaton is more efficient. Garcia and Vidal (Garcia and Vidal 1990) show that the \( k \)-TSSI algorithm is more efficient than the \( k \)-RI algorithm (Angluin 1982).

3.3.4.4 Examples of regular grammar inference

We will present two examples to illustrate the string automaton inference algorithms described above. The first example shows that the \( k \)-RI, KC and \( k \)-TSSI algorithms output the same automaton. The second example shows that the KC and \( k \)-TSSI algorithms are able to work with one example only. As mentioned above, the KC and \( k \)-TSSI algorithms are basically equal algorithms. While the KC algorithm works in terms of automata, the \( k \)-TSSI algorithm works in terms of languages. Below, we will use a characterization based on automata.

Example 3.1 Suppose we have string examples: \( \{aab, abb, ab\} \). The \( k \)-RI and KC algorithms will first build a prefix-tree automaton (PTA), which is shown in the upper part of Figure 3.3, from these examples. Next, both algorithms generalize the PTA by merging some of the nodes or states.

The merging process, using the \( k \)-RI algorithm with \( k = 1 \), is as follows. The automaton in the middle part of Figure 3.3 is the result of merging states 4, 5 and 6 because they satisfy condition 2 in Section 3.3.4.1. Specifically, \( \delta(2, b) \rightarrow 5 \in F \), \( \delta(3, b) \rightarrow 4 \in F \), \( \delta(5, b) \rightarrow 6 \in F \). The automaton in the bottom part of Figure 3.3, which is the final automaton, is the result of merging states 2 and 3 because they satisfy condition 2 in Section 3.3.4.1. Specifically, \( \delta(1, a) \rightarrow 2 \), \( \delta(2, b) \rightarrow 4 \), and \( \delta(2, a) \rightarrow 3 \), \( \delta(3, b) \rightarrow 4 \). The result of the KC algorithm and the \( k \)-TSSI algorithm is the same as the result of the \( k \)-RI algorithm when applied to this set of string examples.

Example 3.2 Now suppose we only have one string example: \( \{aabb\} \). The \( k \)-RI algorithm is not able to generalize beyond the PTA formed by this example. In contrast, the KC and \( k \)-TSSI algorithms using \( k = 1 \), are able generalize beyond the PTA. As shown in Figure 3.4, the occurrence of two equal strings of length 1 irrespective of their next symbols or next states implies that their states are equal. Specifically, \( \delta(1, a) = \delta(2, a) \) and \( \delta(3, b) = \delta(4, b) \). This condition is weaker than condition 2 in Section 3.3.4.1. The resulting automaton represents the language \( a + b^+ \), which is what we might suspect.
CHAPTER 3. IE AND GRAMMATICAL INFERENCE

Figure 3.3: A PTA for example 3.1

Figure 3.4: A PTA for example 3.2
3.3.5 Inference of tree grammars

As previously mentioned, there exist some recent excellent survey papers on the inference of regular grammars, e.g. (Murphy 1996; Sakakibara 1997; Parekh and Honavar 1998). These works survey an extensive number of grammatical inference algorithms that infer finite state automata, context free grammars, and their probabilistic variants. Surveys of tree automata inference methods are rare. One such survey, which is already more than 25 years old, is (Fu and Booth 1975b). Fu and Booth survey three tree automata inference methods, including (Bhargava and Fu 1974) and (Gonzalez and Thomason 1974), that can be considered as the first methods that attempt to infer tree automata. In their book, Gonzalez and Thomason (1978) describe another tree automaton inference algorithm developed in (Gonzalez, Edwards, and Thomason 1976).

Indeed most of the work on grammatical inference try to infer (subset of) regular grammars. One possible reason is that these grammars are simple and have many well understood properties. The problem of learning tree automata is harder than the problem of learning context free grammars (Abe and Mamitsuka 1997; Sakakibara 1997) because tree automata are more expressive than context free automata. As a consequence, the problem of learning tree automata is also harder than the problem of learning regular grammars.

The work on tree automaton inference methods was quite rare, until recently, when work in this domain started to receive more attention. The interest in this domain is driven by, among other things, some recent problems that require the use of automatic knowledge acquisition methods with high expressive power. Examples of these problems are in bioinformatics e.g. predicting protein secondary structure, in natural language processing problems e.g. tackling ambiguity in natural language parsing, and inferring structured documents schema. This is shown by the emergence of several recent works that develop tree automaton inference algorithms.

Grammatical inference can be generalized from string languages to tree languages. Rather than a set of strings over an alphabet $\Sigma$ given as example, we are now given a set of trees over a ranked or unranked alphabet $V$. Rather than inferring a standard finite automaton compatible with the string examples, we now want to infer a compatible tree automaton. The problem can be stated as follows.

**Definition 3.6 (Tree automata inference)**

**Given:** a set of example trees that are in the language $(S^+)$ and a (possibly empty) set of example trees not in the language $(S^-)$,

**Task:** infer a deterministic tree automaton (DTA) $M$ that accepts the trees in $S^+$ and rejects the trees in $S^-$. 


Tree automata are a natural generalization of string automata. As mentioned previously, a string can be viewed as a unary term (nodes with one child). Thus it is quite straightforward to extend the string-based algorithms to work with trees. Typically algorithms for this kind of tree automaton induction are developed by upgrading existing algorithms for string automaton induction. The reversible tree (Besombes and Marion), reversible dependency tree (Besombes and Marion 2001), and function distinguishable tree (Fernau 2002) algorithms are upgrades from the $k$-reversible (Angluin 1982) algorithm. The $k$-testable tree algorithms (Garcia 1993; Knuttila 1993) are upgrades from the $k$-contextual (Muggleton 1990) and $k$-testable string (Garcia and Vidal 1990) algorithms. GIFT (Bernard and de la Higuera 1999) extends the string based method developed in (de la Higuera, Oncina, and Vidal 1996). These tree automata inference algorithms can be categorized as non-probabilistic methods.

In the literature, some probabilistic methods have also been proposed. They are as follows. The probabilistic $k$-testable tree (Rico-Juan, Calera-Rubio, and Carrasco 2000) algorithm extends the $k$-testable tree algorithms (Garcia and Oncina 1993; Knuttila 1993) to output probabilistic $k$-testable tree automata. Both Tlips (Carrasco, Oncina, and Calera-Rubio 2001), an algorithm for learning stochastic tree automata, and an algorithm for inducing stochastic many-sorted tree automata (SMTA) (Habrand, Bernard, and Jacquetet 2002) are upgrades from Algebra (Carrasco and Oncina 1999), which is an algorithm for inducing stochastic deterministic finite automata (DFA). The algorithm to infer stochastic ranked node rewriting grammars (SRNRG) (Abe and Mamitsuka 1997) is an upgrade from the “Inside-Outside” algorithm (Lari and Young 1990) for inducing stochastic context free grammars (SCFGs), which is itself an extension of the Baum-Welch algorithm of the “Forward-Backward” algorithm (Rabiner 1989) for learning Hidden Markov Models (HMMs). Table 3.2 shows a summary of these upgrades.

Table 3.3 shows the characteristics of several tree automaton inference algorithms. Most of the tree automaton inference methods mentioned in Table 3.3 learn from positive examples only, except GIFT (Bernard and de la Higuera 1999). Also note that all tree automaton inference methods mentioned in Table 3.3 are based on ranked trees. To the best of our knowledge, all previous tree automaton inference methods were proposed for ranked trees. Our work in (Kosala, Bruynooghe, van den Bussche, and Blockeel 2003), which proposes a local unranked tree automaton algorithm for IE, is a first step towards algorithms for unranked trees.

3.3.5.1 Inference of $k$-reversible tree automata

The following two algorithms are related to tree automaton inference: RT (Sakakibara 1992), which learns reversible skeletal tree automata and is an extension of the zero reversible (ZR) algorithm (Angluin 1982), and (Garcia
### 3.3 Grammatical Inference

<table>
<thead>
<tr>
<th>String-based methods</th>
<th>Tree-based methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>$k$-reversible (Angluin 1982)</td>
<td>reversible tree (Besombes and Marion 2001), reversible dependency tree (Besombes 2001), function distinguishable tree (Fernau 2002)</td>
</tr>
<tr>
<td>$k$-contextual (Muggleton 1990), $k$-testable string (Garcia and Vidal 1990)</td>
<td>$k$-testable tree (Garcia and Oncina 1993; Knuttila 1993), probabilistic $k$-testable tree (Rico-Juan, Calera-Rubio, and Carrasco 2000)</td>
</tr>
<tr>
<td>(de la Higuera, Oncina, and Vidal 1996)</td>
<td>GIFT (Bernard and de la Higuera 1999)</td>
</tr>
<tr>
<td>Alergia (Carrasco and Oncina 1999)</td>
<td>Tlips (Carrasco, Oncina, and Calera-Rubio 2001), SMTA (Habrard, Bernard, and Jacobnet 2002)</td>
</tr>
<tr>
<td>Hidden Markov Models (HMM) (Rabiner 1989)</td>
<td>SRNRG (Abe and Mamitsuka 1997)</td>
</tr>
</tbody>
</table>

Table 3.2: Some upgrades to tree automaton inference algorithms

and Oncina 1993), which learns recognizable tree sets. These algorithms are not really tree automaton inference algorithms because they learn skeletal tree automata from a set of skeletons or unlabeled derivation trees of a context-free grammar (CFG). It is known that the set of derivation trees of a CFG constitutes a rational set of trees, and can be recognized by some tree automaton. However, reversible skeletal tree automata are basically equal to context-free grammars. Sakakibara (1992) has shown that a class of CFG, called reversible context-free grammars, can be identified in the limit from a positive presentation of structured strings, which are unlabeled derivation trees of an unknown CFG.

We have experimented with a modified version of Sakakibara’s reversible tree (RT) algorithm (Sakakibara 1992). The input to the original RT algorithm is a set of unlabeled derivation trees, while the input to our modified algorithm is a finite set of trees $T$. The modified learning algorithm begins with the trivial partition of the states that accept exactly the examples.

Given an example set $\{a(a(b, c), c)\}$, one obtains the transitions $\{a(2, 3) \rightarrow 1, a(4, 5) \rightarrow 2, c \rightarrow 3, b \rightarrow 4, c \rightarrow 5\}$, and the trivial partition of the states
is \{\{1\}, \{2\}, \{3\}, \{4\}, \{5\}\}. Next, the algorithm generalizes the automaton by merging states. Specifically, it repeatedly merges any two distinct states \(s\) and \(s'\) if any of the following conditions is satisfied:

1. If \(s\) and \(s'\) are both final states.
2. If \(v(q_1, \ldots, q_k) \rightarrow s\) and \(v(q_1, \ldots, q_k) \rightarrow s'\). (Invertibleness)
3. If \(v(q_1, \ldots, q_k, s, q_{k+1}, \ldots, q_l) \rightarrow q\) and \(v(q_1, \ldots, q_k, s', q_{k+1}, \ldots, q_l) \rightarrow q\). (Reset-freeness)

In the above example, we have the transitions \(c \rightarrow 3\) and \(c \rightarrow 5\), hence states 3 and 5 are merged. The algorithm finishes when there is no longer any such pair of blocks and the final partition becomes the output.

Preliminary results with this modified algorithm suggested that it generalises insufficiently on our data sets, which is why we did not pursue this direction further.

Recently, a similar algorithm to our modified algorithm above is proposed in (Besombes and Marion ). Other similar algorithms, that are basically upgrades from the RT algorithm, are recently proposed: reversible dependency tree languages (Besombes and Marion 2001) and function distinguishable tree languages (Fernau 2002).

<table>
<thead>
<tr>
<th>Methods</th>
<th>Need (\oplus) and (\oplus) or (\oplus) only?</th>
<th>Probabilistic?</th>
<th>Merging criterion</th>
</tr>
</thead>
</table>
| reversible tree (Besombes and Marion )  
reversible dependency tree (Besombes and Marion 2001), and function distinguishable tree (Fernau 2002) | \(\oplus\)                                      | no             | deterministic    |
| \(k\)-testable tree (Garcia and Oncina 1993; Knütila 1993)            | \(\oplus\)                                      | no             | deterministic    |
| GIFT (Bernard and de la Higuera 1999)                                   | \(\oplus\) and \(\oplus\)                      | no             | deterministic    |
| Probabilistic \(k\)-testable tree (Rico-Juan, Calera-Rubio, and Carrasco 2000) | \(\oplus\)                                      | yes            | deterministic    |
| Tlips (Carrasco, Oncina, and Calera-Rubio 2001)                        | \(\oplus\)                                      | yes            | probabilistic    |
| SMTA (Habrard, Bernard, and Jacquetnet 2002)                            | \(\oplus\)                                      | yes            | probabilistic    |
| SRNRG (Abe and Mamitsuka 1997)                                          | \(\oplus\)                                      | yes            | deterministic    |
3.3. GRAMMATICAL INFECTION

3.3.5.2 Inference of k-testable tree automata

The $k$-contextual (Muggleton 1990) and $k$-testable string (Garcia and Vidal 1990) languages are characterized by the substrings of length $k$ (also called $k$-grams) in the string examples. Indeed, (the probabilistic versions of) these languages can be regarded as the $k$-gram models that have been widely used in the natural language processing community (Brown, Pietra, deSouza, Lai, and Mercer 1992; Jelinek 1998).

The $k$-testable tree (Garcia 1993; Knuttila 1993) languages are the natural extension of the $k$-contextual (Muggleton 1990) and $k$-testable string (Garcia and Vidal 1990) languages. If in the string languages we have $k$-grams, the $k$-testable tree languages are characterized by $k$-subtrees. Loosely speaking, $k$-subtrees are all the subtrees of height $k$ in a tree.

In (Garcia and Oncina 1993; Knuttila 1993), the $k$-subtrees in the examples are divided into three types to create an automaton from the tree examples. They are $k$-root subtrees, $k$-fork subtrees, and $k$-subtrees. The $k$-root subtrees, $k$-fork subtrees, and $k$-subtrees are the subtrees that are located at the top, middle, and bottom part of a tree. Then the transitions are created from these three types of subtrees. In the $k$-testable tree languages, two equal subtrees always have the same state.

The tree automaton inference algorithms that are described in this thesis are based on the $k$-testable tree languages. A more detailed and formal description of the $k$-testable tree automaton inference algorithm is given in Chapter 4.

The $k$-testable tree automaton inference algorithm has some useful properties as stated in (Rico-Juan, Calera-Rubio, and Carrasco 2000):

- it runs in time polynomial to the number of nodes in the tree examples,

- the resulting tree automaton can be updated incrementally,

- it allows for a smaller generalization degree (when using $k > 2$) than tree automata obtained from the generalization of the trivial partition of the samples (a case that corresponds to $k = 2$),

- it requires a relatively small number of examples compared to other tree-based algorithms that use state merging methods.

Besides these advantages, the $k$-testable tree languages (also called locally testable tree languages in the strict sense) have the disadvantage that they are not expressive enough to capture long-term constraints or dependencies. Hence, the family of locally testable tree languages in the strict sense are only suitable in the situations or problems where the dependencies are local.
3.3.5.3 Inference of probabilistic tree automata

Simply said, a probabilistic grammar is a grammar with probabilities associated with its transition rules. The problem of learning probabilistic or stochastic grammars from examples has two aspects (Sakakibara 1997):

- determining the structure or topology of the grammar, and
- estimating probabilistic parameters in the grammar.

With respect to these aspects, we can differentiate between methods that learn the structure and estimate the parameters separately and methods that learn the structure and estimate the parameters simultaneously. Some examples of the former are (Abe and Manisaka 1997; Rico-Juan, Calera-Rubio, and Carrasco 2000). Some examples of the latter are (Carrasco, Oncina, and Calera-Rubio 2001; Habrard, Bernard, and Jacquenet 2002; Stokke and Omohundro 1994). In the literature, the problem to find the structure of the grammar is known to be the harder problem of the two.

A typical approach used by the methods that learn the structure and estimate the parameters separately is to start with a fully connected HMM or SCFG, and to run the parameter estimation algorithms to get a locally maximal estimation of probabilities. Then the structure of the grammar is obtained by removing transitions with zero or low probabilities. Thus the merging between states is done deterministically. On the other hand, the methods that learn the structure and the parameters estimation simultaneously typically merge states based on a probabilistic or statistical criterion.

If one learns from “bracketed” examples, which explicitly contain structural information of the derivation tree, then the process of learning stochastic grammars can be simplified. This is because the learner does not have to implicitly consider all possible derivations of the training strings when reestimating the grammar’s parameters. The structure contained in the bracketed examples gives an initial estimation of the structure of the grammars.

**Definition 3.7** A stochastic deterministic tree automaton (SDTA) is defined as $M = (V, Q, \delta, p, r)$ where $V$ is a finite set of labels, $Q$ is a finite set of states, $\delta : V \times Q^* \to Q$ is the transition relation, $p : V \times Q^* \to [0, 1]$ is the probability of a transition. The function $r : Q \to [0, 1]$ gives the probability that a state $q \in Q$ is an accepting state. To be precise, one should define the state of a tree (or subtree) $t$ as follows: $\delta(t) = \delta(v)$, if $t = v \in V$ and $\delta(t) = \delta(v, \delta(t_1), \ldots, \delta(t_n))$, if $t = v(t_1, \ldots, t_n) \in T_v$

A SDTA parses the tree bottom-up. A state and a probability are associated with each node of the tree. The labeling of each node is defined by the transition relation. The probability of a tree $t$ in the language generated by $M$ is given
by the product of the probabilities of all transitions used when \( t \) is processed by \( M \) times \( r(\delta(t)) \):

\[
p(t|M) = r(\delta(t)) \times \pi(t)
\]

where \( \pi(v(t_1, \ldots, t_n)) \) is recursively given by:

\[
\pi(t) = p(v, \delta(t_1), \ldots, \delta(t_n)) \times \pi(t_1) \times \ldots \times \pi(t_n)
\]

A tree is accepted if its probability, that is \( p(t|M) \), is strictly positive. An SDTA defines a probability distribution over all trees built using a certain alphabet that are accepted by this automaton. To produce a consistent SDTA, the following normalizing constraint must hold:

\[
\sum_{t \in \tau} p(t | M) = 1
\]

### 3.3.5.4 Inference of probabilistic \( k \)-testable tree automata

In (Rico-Juan, Calera-Rubio, and Carrasco 2000), the \( k \)-testable tree inference algorithms (Garcia 1993; Knuutila 1993) are extended to output a probabilistic \( k \)-testable tree automaton. First the algorithm infers a non-probabilistic \( k \)-testable tree automaton using an algorithm similar to the algorithms proposed in (Garcia 1993; Knuutila 1993). That is by collecting the \( k \)-root subtrees, \( k \)-fork subtrees, and \( k \)-subtrees. After that the transitions are produced from these subtrees.

The algorithm computes the probabilities of the transitions by counting the occurrences of the subtrees in the training examples. The following example describes how the probabilities are calculated. Suppose the method is given a set of stochastic tree samples \( S = \{ \tau_1, \tau_2, \ldots, \tau_n \} \); the probabilities of accepting states \( r(t) \) are estimated from \( S \) by the following formula:

\[
r(t) = \frac{1}{n} \sum_{i=1}^{n} \text{equalroot}(t, \tau_i)
\]

with \( \text{equalroot}(t, \tau_i) = 1 \) if \( t \) appears at the root of the tree sample \( \tau_i \) and zero otherwise; and the transition probabilities \( p \) of all subtrees are calculated as follows:

\[
p(t) = \frac{\sum_{i=1}^{n} C(t, \tau_i)}{\sum_{i=1}^{n} C(\text{root}(t), \tau_i)}
\]

where \( C(t, \tau) \) counts the number of subtrees \( t \) in \( \tau \) and function \( \text{root}(t) \) outputs the root of subtree \( t \). In (Rico-Juan, Calera-Rubio, and Carrasco 2000), if \( \text{height}(t) = k \) then \( \text{root}(t) \) is the subtree \( t' \) in the upper part of \( t \) with \( \text{height}(t') = k - 1 \). Suppose \( t = a(b, c(d, e(f))) \) then \( \text{root}(t) = a(b, c, e) \).
3.3.5.5 Inference of stochastic deterministic tree automata

Carrasco, et al. (Carrasco, Oncina, and Calera-Rubio 2001) describe the algorithm Tips that infers stochastic deterministic tree automata (SDTA) (see Section 3.3.5.3). The algorithm works bottom-up by starting with a trivial partition before merging some states in the partition. Given a set of training examples of trees, Tips starts with an ordered set of all subtrees from the training examples. These subtrees are ordered by their height and then they are mapped to some states. Then it iterates by computing the similarity between two subtrees using a statistical test that depends on a parameter \( \alpha : [0, 1] \). Intuitively \( \alpha \) directs the sensitivity whether any two states should be merged. The probabilities of the transitions used to calculate the similarity are computed by counting the occurrence of the subtrees in the training examples.

Algorithm Tips has been shown to be able to learn a consistent automaton in the limit of Gold and requires a run time polynomial in the number of subtrees in the training examples.

3.3.5.6 Inference of stochastic many-sorted tree automata

An extension of the Tips algorithm that takes sorts into account to learn a stochastic many-sorted tree automaton (SMTA) is proposed in (Habrand, Bernard, and Jacquetet 2002).

**Definition 3.8** A signature \( \Sigma \) is a quadruple \((S, X, \alpha, \sigma)\), where \( S \) is a finite set whose elements are called sorts, \( X \) is a finite set whose elements are called function symbols, \( \alpha \) is a mapping from \( X \) to a natural number, \( \alpha(v) \) is called the arity of \( v \), \( \sigma \) is a mapping from \( X \) to \( S \), and \( \sigma(s) \) is called the sort of \( s \).

Informally, a signature is the set of all symbols that appear in the terms, including their arity and sort. A sort is also called a type of the symbol appearing in the set of terms. A symbol \( a \) appears at position \((v, n)\) if it is the \( n^{th} \) argument of a term constructed on the symbol \( v \). Two symbols are of the same sort if they appear at least once at the same position.

**Definition 3.9** An SMTA is a quintuple \((\Sigma, Q, \delta, p, r)\), where \( \Sigma \) is a signature \((S, X, \alpha, \sigma)\), \( Q = \bigcup_{s \in S} \mathbb{Q}^s \) is a finite set of states with each state having a sort in \( S \), \( r : Q \rightarrow [0, 1] \) is the probability that a state is an accepting state, \( \delta : V \times Q^* \rightarrow Q \) is the transition relation, \( p : V \times Q^* \rightarrow [0, 1] \) is the probability of a transition.

The parsing strategy, the labeling function, and the acceptability of an SMTA are similar to those of the stochastic deterministic tree automaton (SDTA) described above in Section 3.3.5.3. The probability of a tree \( t \) in the language can be calculated using the same formula as for the SDTA.
3.3 GRAMMATICAL INFERENCE

The learning algorithm to learn an SMTA is similar to Tlips. It uses the same statistical test as the one used in Tlips to compute the similarity between two subtrees. The difference is that it only computes the similarity of subtrees built from symbols of the same sort because it learns from many-sorted trees.

The resulting SMTA can be generalized further by using a kind of wildcard inside some similar transitions. For example, given the following three transitions: \( v(q_1, q_2, q_3) \rightarrow q, v(q_1, q_4, q_2) \rightarrow q, \) and \( v(q_5, q_6, q_3) \rightarrow q. \) A possible generalization of these transitions is \( v(q_1, s, q_3) \rightarrow q. \) This generalization is searched by traversing the lattice of terms for the maximally specific general rule having a score greater than \( \gamma : [0, 1]. \)

While non-probabilistic automata are typically evaluated based on the accuracy, probabilistic automata are evaluated based on their ability to correctly predict the probability of a test sample. One such measure is the perplexity, which is a kind of measure that indicates the difference between the distribution of the unknown target to the distribution of the training set. The use of sorts to learn the SMTA has been shown to reduce the perplexity significantly (Habrard, Bernard, and Jacquenet 2002). The generalization of SMTA is also shown to improve the predictive power of the automaton; it requires less examples to converge.

3.3.5.7 Inference of stochastic ranked node rewriting grammars

An algorithm that learns stochastic ranked node rewriting grammars (SRNRG) for predicting protein secondary structures was proposed in (Abe and Mamitsuka 1997). The ranked node rewriting grammars (RNRG) are members of the family of tree grammars (Abe and Mamitsuka 1997). Thus they are tree generating systems. A RNRG consists of a single tree called the starting tree and a finite number of rewriting rules that rewrite a node in a tree with an incomplete tree (or a subtree). The application of any rewrite rule to a tree node follows the following conditions:

1. Only a node that is labeled with a non-terminal symbol can be rewritten.

2. the rank (number of children) of a node that is being rewritten has to be equal to the number of "empty nodes" in the incomplete tree. After rewriting, the children of the rewritten node are attached to these empty nodes.

The trees generated by an RNRG are the set of all trees whose nodes are solely labeled with terminal symbols that can be generated from the starting tree by a finite number of applications of its rewrite rules.

The algorithm to infer stochastic ranked node rewriting grammars (SRNRG) (Abe and Mamitsuka 1997) is an upgrade from the Inside-Outside algorithm (Lari and Young 1990) for inducing stochastic context free grammars (SCFG),
which is itself an extension of the Baum-Welch algorithm or the Forward-Backward algorithm (Rabiner 1989) for learning Hidden Markov Models. The Forward-Backward algorithm and the Inside-Outside algorithm are iterative procedures that can be used to estimate the probabilistic parameters of an HMM and SCFG respectively. They are based on the expectation-maximization (EM) technique that increases the likelihood of the training set in each step until a local maximum is reached. Given a set of positive examples and a SCFG with randomly initialized parameters, the Inside-Outside algorithm first computes the most probable parse tree for each training sentence. The derivations are then used to reestimate the probabilities associated with each rule and the procedure is repeated until the probabilistic values stabilize.

The ranked node rewriting grammars are argued to be suitable to model the long distance dependencies of some corresponding β-sheet regions in an amino acid sequence (Abe and Mamitsuka 1997). Some types of dependencies present in an amino acid sequence, are parallel, anti-parallel, and combination of parallel and anti-parallel dependencies. These types of dependencies cannot be captured by context free grammars and require more powerful types of grammars, such as tree grammars. In (Abe and Mamitsuka 1997), an SRNRG trained by data from a particular protein was able to predict the location and structure of β-sheets in the test sequences of different proteins that had only a small structural similarity with the training sequences.

3.3.5.8 GIFT: inference of logical terms

GIFT (Bernard and de la Higuera 1999) is a system that learns logical recursive rules from a set of positive and negative examples and background knowledge. Because it learns logical first-order rules, GIFT can be categorized as an inductive logic programming (ILP) technique. At the heart of GIFT is a grammatical inference algorithm, called MDTAmfer, that learns many-sorted deterministic tree automata from positive and negative examples.

GIFT requires examples that are in the form of a logical interpretation constructed by a human expert. These logical interpretations are given in the form of tree-structured terms. These tree examples are transformed into many-sorted terms through a type inference algorithm. Both positive and negative examples, in the form of many-sorted terms, are then given as input to MDTAmfer, which will output a tree automaton. Finally, an algorithm converts the tree automaton into a logic program.

Compared to the other tree automaton inference techniques mentioned above, this system is unique in the sense that a tree automaton inference method is used to learn a logical program and that it uses both positive and negative examples. This is different from the other tree approaches.
3.4 Summary

We have started the chapter with a brief overview of current IE research. Given the size of the field, we have restricted ourselves to giving a brief overview of IE. In Chapter 6, we will describe a more thorough review of the current state-of-the-art in structured IE methods, because they are most relevant to our work.

In the second part of this chapter we have briefly introduced the grammatical inference problem. Next, we have described the setting of learning from positive examples only, which is the setting that we use for our tree automaton inference methods. Then, we have described three algorithms that infer regular grammars. They are the bases of the tree automaton inference algorithms that we use and develop in this thesis. Finally, we have given two examples illustrating how these three regular grammar inference algorithms work.

We have also reviewed several tree automaton inference methods in this second part, including the $k$-testable tree automaton inference algorithm that becomes the basis of our tree automaton inference methods. One can say that tree automaton inference is a hot topic at the moment. This is shown by the number of algorithms proposed in the last three years. References to other tree automaton inference algorithms can be found in (Besombes and Marion; Fernau 2002).

Out of the available ranked tree automaton inference algorithms, we have chosen to use the non-probabilistic tree automaton inference methods because they are simpler and easier to implement. The most important benefit of the non-probabilistic methods is that they typically require less training data and thus are faster to learn. This characteristic is very important for developing practical IE systems. Indeed, it allows to minimize the users’ effort in labeling examples.

In this chapter, we did not describe how the tree grammar inference can be used for information extraction. This issue will be discussed in the next chapter.
Chapter 4

IE by tree automaton inference

4.1 Introduction

In Chapter 1, we have described our motivation for the work in this thesis. In this chapter, we start by describing our motivation for tree-based IE further. In the previous chapter, we have described information extraction problems and grammatical inference techniques separately, and in this chapter we describe how the grammatical inference techniques can be used for information extraction problems. After this, we describe two practical issues that emerge when applying tree grammar inference on structured documents. Then we present some characteristics of our approach.

The rest of this chapter consists of two main parts. In the first part, we describe our methods of information extraction using three ranked tree automaton inference algorithms: k-testable, g-testable, and gl-testable. The k-testable algorithm was originally described in (Garcia 1993; Knuutila 1993; Rico-Juan, Calera-Rubio, and Carrasco 2000). The other two algorithms, g-testable (Kosala, Bruynooghe, Blockeel, and Van den Bussche 2002) and gl-testable (Kosala, Bruynooghe, Van den Bussche, and Blockeel 2002), are our extensions of the k-testable algorithm. In the second part, we present our methods of information extraction using an unranked tree automaton inference algorithm, introduced in (Kosala, Bruynooghe, Van den Bussche, and Blockeel 2003), which is so far a less explored alternative.
4.2 From string-based to tree-based IE

As mentioned previously, the focus of this thesis is work on IE from structured texts. A characteristic of structured documents is that they contain implicit structures. In this thesis, we exploit these structures for information extraction and investigate if the structures improve the extraction performance. Below we would like to reemphasize our motivation.

4.2.1 String-based IE

Many structural IE systems (wrappers) have been applied to structured data, these include (Hsu and Dung 1998; Freitag and McCallum 1999; Kushnerick, Wek, and Doorenbos 1997; Kushnerick 2000; Muslea, Minton, and Knoblock 2001; Freitag and Kushnerick 2000; Chidlovskii, Ragetti, and de Rijke 2000). These systems can be seen as using machine learning or grammatical inference techniques to induce a kind of delimiter-based string patterns. However, these methods consider structured document to be a string, not a tree. The training process employed by these string-based methods is shown in the upper part of Figure 4.1.

Structured documents, such as HTML or XML documents, contain tags. In string-based approaches, these tags are viewed as parts of the string representing the document. The training examples are provided in the following way: the fields of interest in a document are annotated or labeled with a special marker or tag that is different from the texts or tags found in the training set. Then, depending on the method, some pre-processing steps might be needed, such as splitting the document into small fragments, and selecting some of them for use as training examples, e.g. (Soderland 1999); specifying manually the length of a window for the prefix, suffix and target fragments (Freitag and McCallum 1999; Freitag and Kushnerick 2000), and dealing with special tokens or landmarks such as ‘>’ or ‘;’ (Freitag and Kushnerick 2000; Muslea, Minton, and Knoblock 2001). Finally, these examples are used as input for the learning algorithm.

4.2.2 Tree-based IE

Structured documents such as HTML and XML documents, however, have a tree structure. Thus it is a natural extension to consider utilizing the tree structure to extract structured documents. Actually, the idea of utilizing the tree structure of structured documents is not new. There have been several previous works that suggest using the (partial or simplified) tree structure of structured documents, for instance (Hsu and Yih 1997; Cohen and Fan 1999; Liu, Pu, and Han 2000). More recent works that follow this idea are,
4.2. FROM STRING-BASED TO TREE-BASED IE

![Diagram of tree-based IE]

Figure 4.1: From string-based to tree-based IE

e.g. (Cohen, Hurst, and Jensen 2002; Sakamoto, Arimura, and Arikawa 2002; Hennani and Bressan 2002).

In the following discussions, we will only give a brief description of each of these tree-based approaches. A more detailed description is given in Chapter 6. (Hsu and Yih 1997) cast the IE problem into a tree matching problem between a template tree and the HTML document tree. (Cohen and Fan 1999) use several learning algorithms, such as RIPPER, CART, etc., to learn page-independent wrapper heuristics and page-specific wrappers. XWrap (Liu, Pu, and Han 2000) is a wrapper generator system, in which the user uses an HTML parse tree to specify the wrapper. WLL2 (Cohen, Hurst, and Jensen 2002) is a logic-based wrapper learner that uses multiple representations of HTML documents, such as string, tree, visual, and geometric representations. Sakamoto et al. (Sakamoto, Arimura, and Arikawa 2002) propose a certain class of wrappers. Their wrappers work in a similar way to the way XPath (XPath 1999) expressions work. Hennani and Bressan (Hennani and Bressan 2002) propose a tree alignment algorithm that is based on two heuristics for extracting multiple records from web documents.

This thesis proposes and explores several alternatives to tree-based IE. Our methods stem from the fact that tree automata are well-established and natural
tools for processing trees (Comon, Dauchet, Gilleron, Jacquemard, Lugiez, Tison, and Tommasi 1999). Therefore the use of tree automata for IE from structured documents is well-motivated. Interestingly, a recent work by Gottlob and Koch (Gottlob and Koch 2002) shows that all existing wrapper languages for structured document IE can be captured using tree automata, which justifies our approach further. The use of tree automata to extract information from structured web documents is a new idea.

Some characteristics of our tree automata based methods, that differentiate them from previous tree-based approaches that employ tree structures, are as follows:

- Besides acting as a wrapper, a tree automaton also reflects the grammar of a set of HTML or XML documents in the training examples, because a tree automaton accepts or rejects the whole document tree.

- A tree automaton can capture extraction problems where the target fields are dependent on some structural (also called contextual) information.

- Our methods are rooted in a grammatical inference setting and learn from positive examples only.

- Our methods require little user intervention, which is mainly for labeling examples.

4.3 IE by grammatical inference

Following Freitag (Freitag 1997), we can map an IE task into a grammar inference task as follows.

**Definition 4.1 (Information extraction as string grammar inference)**

*Given:* A set of documents, possibly from the same domain or topic. Each document is preprocessed into a sequence of tokens, from an alphabet \( \Sigma \). In a set of documents, the fields to be extracted are replaced by a special token \( z \).

*Task:* Infer a DFA for a language \( L \subseteq (\Sigma \cup \{z\})^* \), that accepts training examples, where the fields to be extracted are replaced by \( z \). The inferred automaton functions as a wrapper or extractor for the fragments of interest from a set of “similar” documents.

The term “similar” is used in an empirical sense, which might mean web pages from the same site or the same database, such as online bookstores,
white pages, etc. It is straightforward to extend the above definition to tree
grammar inference. Then the IE task consists of selecting certain nodes from
a tree. We are given a set of examples, each consisting of a tree and a selected
node. By adding for each label $v \in V$ a new label $(v, x)$, where $x$ is a new
"target" symbol, we can represent such examples as trees over the new alphabet
$V' = V \cup (V \times \{x\})$, where the label of precisely one node, namely the selected
one, is in $V \times \{x\}$, and the other labels are in $V$ as before.

We can now try to infer a grammar for the obtained set of example trees,
producing a DTA $M$ over $V'$. If successful, we can use $M$ to perform the original
IE task simply by selecting each node, one by one, relabeling it to $(v, x)$ if its
original label is $v$, and verifying whether $M$ accepts the thus relabeled tree. If
so, the selected node is extracted.

In what follows, we will focus on applications where only leaf nodes are to be
extracted, and where all these nodes have a fixed known label. We can therefore
simplify the setting a bit by labeling selected nodes simply by the target symbol
$x$ instead of $(v, x)$, because $v$ is fixed and known and therefore uninformative.
The new alphabet $V'$ then simply becomes $V \cup \{x\}$. The problem of tree
automaton-based IE can be summarized in the following definition.

**Definition 4.2 (Information extraction as tree grammar inference)**

**Given:** A set of structured documents, possibly from the same domain or
topic. Each document is preprocessed into a tree, from an alphabet $V$.
In a set of documents, one of the fields to be extracted is replaced by a
special token $x$.

**Task:** Infer a tree automaton for a language $L \subseteq T(V \cup \{x\})$ that accepts tree
examples, where one of the fields to be extracted (a leaf node) is replaced
by $x$. The inferred automaton functions as a wrapper or extractor of the
fragments of interest from a set of "similar" documents.

When using the learned automaton, a similar transformation is done. Each
leaf that is a candidate for extraction is in turn replaced by $x$. The token
replaced by $x$ is extracted iff the transformed document is accepted by the
learned automaton.

### 4.4 Approach and issues

Structured documents in HTML, or more generally, XML formats, can be
readily represented as trees, where internal nodes represent the elements, and
are labeled by tags, and leaf nodes represent the text content. Before we can
use grammatical inference to perform IE on such trees, as described above, we
must deal with two issues:
1. How do we deal with text content?

2. Tags are not ranked. For example, in HTML, an `<ul>` element can have an arbitrary number of `<li>` subelements, and more generally, in XML documents, there is no bound on the number of subelements an element can have.

In the next two subsections, we will deal with these two issues. After that, we will describe several characteristics of our approach.

### 4.4.1 Preprocessing text content

Figure 4.2 shows a simplified view of a representative document. The real documents, as used in the experiments by us and the other authors, are more complex. In this document, the fields to be extracted are the fields following the "Alt. Name" and "Organization" fields. A document consists of a variable number of records. As we can see, in each record the number of occurrences of the fields to be extracted is also variable (from zero to several occurrences). Also the position where they occur is not fixed. There is evidence that extracting this kind of information is a difficult task (Hsu and Chang 1999; Muslea, Minton, and Knoblock 1999).

An important issue is how to deal with the various text nodes in the document. Treating every piece of text as a distinct label is unacceptable as it results in too specific automata. Labeling all text nodes (but the node to be extracted which is labeled x) by some fixed label CDATA, as in XML DTD's (XML 2000), is also unacceptable, as this results in too general automata.

Indeed, consider the following fragment of a document tree, which is shown in Figure 4.3, that originated from the document shown in Figure 4.2. Suppose the target field x is always preceded by a field labeled Organization. If the labels Provider and Organization are both replaced by CDATA then any automaton that extracts the x node will likely also extract the att global services node when it is replaced by x. Hence we should not replace the field Organization by CDATA. Fields such as Organization and Alt.Name are called distinguishing contexts (or structural contexts). Roughly speaking, a distinguishing context is the text content of a tree node that is useful for the identification of the field of interest. However not every field of interest has a unique distinguishing context.

In our experiments, we consistently used the following procedure to determine the distinguishing context. We look for the invariant text label that is nearest to the field of interest and occurs at the same distance from the field of interest in all examples. For example, the text "Organization" is an invariant text label that is nearest to the organization name in HTML document figure at the beginning of this Section. If no such text is found, no context is used and
4.4. APPROACH AND ISSUES

Figure 4.2: An example of an HTML document
all text is turned into CDATA. If there are several possibilities, one is chosen at random. As distance measure, we use the length of the shortest path in the document tree (for example the distance of a node to its parent is one; to its sibling, two; to its uncle, three). The above procedure has been implemented and validated. Thus, this process does not add extra work for the user.

4.4.2 Two ways to model tree structured documents

In the literature, existing tree automaton inference algorithms expect ranked trees. Thus, a problem when directly applying existing tree automaton inference algorithms to tree-structured documents such as HTML or XML documents, is that the latter trees are “unranked”: the number of children of a node is not fixed by the label, but is varying. There are two approaches to deal with this situation:

1. The first approach is to use a generalized notion of tree automata towards unranked tree automata formalisms (e.g., (Pair and Quere 1968; Takahashi 1975)). In such formalisms, the transition rules are of the form $\delta(v,e) \rightarrow q$, where $e$ is a regular expression over $Q$ that describes a sequence of states.

2. The second approach is to encode unranked trees into ranked trees, specifically, binary trees. Thus we can use existing tree automaton inference algorithms for inducing the tree automaton.

In this thesis, we explore both approaches. The advantages and the disadvantages of the two approaches are as follows.
4.4. APPROACH AND ISSUES

- The advantage of the first approach:
  - The trees do not need to be converted to ranked trees. Thus, we save computational effort. More importantly, we preserve the structure and the distance from the target field to the context.

- The disadvantage of the first approach:
  - This is so far a less explored alternative and algorithms for inducing unranked tree automaton do not exist yet. Thus we cannot use existing algorithms, such as those described in Chapter 3, that have been developed for ranked trees.

- The advantage of the second approach:
  - It is less complicated because we can directly use the tree automaton inference algorithms that have been proposed in the literature, such as those described in Chapter 3.

- The disadvantages of the second approach:
  - We have to convert unranked trees to ranked trees before applying the algorithm.
  - More importantly, the distance between the distinguishing context and the target field might increase because the conversion to ranked trees increases the height of the trees.

These two approaches will be elaborated in the later sections in this chapter, where we will also introduce several concrete grammatical inference algorithms that we use for IE from structured documents.

4.4.3 Some characteristics

Our approach for information extraction has the following characteristics:

- Strings stored at the nodes are treated as a whole. If extracted, the whole node is returned. For example, our method is able to extract the whole node “att global services” in Figure 4.3, but is not able to extract the substring “att” only.

- One automaton is learned for one type of field to be extracted, e.g., the field following “Organization” in Figure 4.3.

- In the examples used during learning, one target field is replaced by x. When a document contains several fields of the same type, then several examples are created from it, one for each occurrence of the target field.
One characteristic of our tree automata wrappers is that they do single-slot (or single-field) extraction. A single-slot IE system extracts isolated facts from the text, while a multi-slot IE system groups the related extracted fields together into correctly ordered multi-slot facts. A group of related extracted fields is commonly called a case frame. There are some domains where multi-slot extraction is a necessity. For example, a webpage may contain a list of house addresses with their corresponding prices. Unless the address and the price are combined in a pair, the extracted information is rather useless because we do not know which price should be the correct attribute for a particular address.

Multi-slot extraction can be achieved by extracting also the location of the extracted fields. Knowing the locations, one can combine the extracted fields into the correct tuples. This simple post-processing method works for structured documents such as HTML/XML documents, since the order of the fields in a case frame typically follows the order of their position in the document. However, we did not collect the extracted fields in a case frame in our experiments.

4.5 IE with ranked tree automaton inference algorithms

4.5.1 Introduction

In this section, we explore the application of the $k$-testable tree automaton inference algorithm, which was developed in (Garcia 1993; Knuttila 1993; Rico-Juan, Calera-Rubio, and Carrasco 2000). We develop a novel wrapper induction method that utilizes the tree structure of the document and uses $k$-testable tree automata as wrappers. Informally, a $k$-testable tree language is a language that can be determined just by looking at all the subtrees of height $k$.

As mentioned previously in the previous chapter, we choose the $k$-testable algorithm because it requires relatively less examples than other tree-based algorithms. This characteristic is important for learning a practical wrapper because the user effort to provide labeled examples should be minimized.

The $k$-testable tree automaton inference algorithm (Garcia 1993; Knuttila 1993; Rico-Juan, Calera-Rubio, and Carrasco 2000) is a grammatical inference algorithm that is able to identify in the limit (Gold 1967) any $k$-testable tree language in the strict sense from positive examples only. The amount of generalization occurring when learning from the positive examples of the $k$-testable tree algorithm is mainly determined by the value of $k$; it decreases with increasing $k$. It is an algorithm for ranked trees, while documents are unranked
trees. Thus, we have to convert web documents into (ranked) binary trees to apply the algorithm.

As the extraction is based on some structural context, $k$ must be large enough such that the field to be extracted and its structural context are covered in the same subtree. Because of the binarization, the value of $k$ needed for capturing the structural or distinguishing context tends to be rather large. Consequently, the generalization tends to be rather low, often resulting in rather poor recall, as we reported in (Kosala, Van den Bussche, Bruynooghe, and Blockeel 2002).

To overcome this problem, we have proposed two generalizations of the $k$-testable algorithm namely $g$-testable and $gl$-testable algorithms. In the $g$-testable algorithm (Kosala, Bruynooghe, Blockeel, and Van den Bussche 2002), the generalization is parameterized by $l$. It considers generalizations of states, which are trees, where the state labels at the lowest $l$ levels are replaced by wildcards. The $gl$-algorithm, which is introduced in (Kosala, Bruynooghe, Van den Bussche, and Blockeel 2002; Kosala, Blockeel, Bruynooghe, and Van den Bussche 2002), considers another generalization and uses the partial order between different generalizations to limit the search. Experiments show that these generalizations improve the performance of the induced wrappers.

### 4.5.2 Training and testing process

The learning procedure is as follows (see also Figure 4.1):

1. Replace in the examples the target field by ‘$x$’, the distinguishing context (if present) by ‘ctx’ and all other text fields by CDATA.
2. Convert the example trees to binary trees.
3. Run a ranked tree automaton inference algorithm on the examples and return the inferred automaton.

The extraction procedure, which is quite similar to the learning procedure, is as follows:

1. Replace the distinguishing context (if present) by ‘ctx’ and all other text fields by CDATA.
2. Convert the tree to a binary tree.
3. Repeat for all CDATA nodes:
   - Replace the label of one CDATA node by the special label ‘$x$’.
   - Run the inferred tree automaton.
• If the tree is accepted by the automaton, then extract the original
text of the node labeled with $x$.

The automaton can succeed for zero, one or more text nodes. The text
nodes for which it succeeds are the extracted fields. Only the first step of the
learning procedure requires user intervention. The second step of the learning
procedure and the whole extraction procedure are done automatically. The
above procedures are repeated for each field of interest in the dataset. If one
wants to do a novel extraction task on a novel dataset, then the learning pro-
cedure above has to be done for this novel task. That is the user should mark
the fields of interest and distinguishing context if it exists, then run the learning
algorithm on the novel data.

4.5.3 Conversion to ranked trees

Existing tree automata inference algorithms expect ranked trees. The simplest
way to apply them on HTML or XML documents, which are unranked trees,
is to transform the latter into (ranked) binary trees. This is the approach that
we follow in this section.

Using the symbol $T$ to denote unranked trees and $F$ to denote a sequence
of unranked trees (a forest), the following grammar defines unranked trees:

$$T ::= a(F), a \in V$$
$$F ::= \epsilon$$
$$F ::= T, F$$

The transformation we use can be formally defined with the following re-
cursive function $\text{encode}$ (with $\text{encode}_f$ for the encoding of forests):

$$\text{encode}(T) \equiv \text{encode}_f(T, \epsilon)$$
$$\text{encode}_f(a(F_1), F_2) \equiv \begin{cases} 
    a & \text{if } F_1 = F_2 = \epsilon \\
    a_{\text{left}}(\text{encode}_f(F_2)) & \text{if } F_1 = \epsilon, F_2 \neq \epsilon \\
    a_{\text{right}}(\text{encode}_f(F_1)) & \text{if } F_1 \neq \epsilon, F_2 = \epsilon \\
    a(\text{encode}_f(F_1), \text{encode}_f(F_2)) & \text{otherwise}
\end{cases}$$

Informally, the first child of a node $v$ in an unranked tree $T$ is encoded as
the left child of the corresponding node $v'$ of $T'$, the binary encoding of $T$,
while the right sibling of a node $v$ in tree $T$ is encoded as the right child of $v'$
in $T'$. To distinguish between a node with one left child and a node with one right
child, the node is annotated with left and right respectively. For example, the
unranked tree $a(b, c(a), d)$ is encoded into the binary tree $a_{\text{left}}(b_{\text{right}}(c(a, d)))$
as pictorially shown in Figure 4.4 and the detail of the encoding steps is shown
in Figure 4.5. Note that the binary tree has exactly the same number of nodes
as the original tree.
4.5 IE WITH RANKED TREE ALGORITHMS

![Diagram of a rooted tree and its encoding steps]

Figure 4.4: An example of conversion from unranked into binary tree.

```
1: encode(a(b, c(a), d))
2: encode_f(a(b, c(a), d), c)
3: a_{left}(encode_f(b, c(a), d))
4: a_{left}(b_{right}(encode_f(c(a), d)))
5: a_{left}(b_{right}(c(encode_f(a), encode_f(d))))
6: a_{left}(b_{right}(c(a, d)))
```

Figure 4.5: Encoding steps

### 4.5.4 Definitions

The \( k \)-testable algorithm is a basic tree automaton inference algorithm. As we will see in Chapter 5, the algorithm is precise, but is sometimes too specific, as indicated by the low recall. Hence we develop two generalisation algorithms: \( g \)-testable and \( g_l \)-testable. We start with some definitions.

With \( t \) a tree, height \( (t) \) is the number of nodes on the longest path from the root. The \( k \)-root \( r_k(t) \) of a tree \( t \) is the tree of height at most \( k \) obtained from \( t \) by cutting off branches longer than \( k \); the set \( f_k(t) \) of \( k \)-forks is the set of all trees of height \( k \) obtained from \( t \) by taking all subtrees of height at least \( k \) and cutting off branches longer than \( k \); finally the set \( s_k(t) \) of \( k \)-subtrees is the set of all subtrees at the bottom of \( t \) of height at most \( k \). Formally:

\[
r_k(v(t_1, \ldots, t_m)) = \begin{cases} 
  v & \text{if } k = 1 \\
  v(r_{k-1}(t_1), \ldots, r_{k-1}(t_m)) & \text{if } k > 1 
\end{cases}
\tag{4.1}
\]

\[
f_k(v(t_1, \ldots, t_m)) = \begin{cases} 
  \emptyset & \text{if } \text{height}(v(t_1, \ldots, t_m)) < k \\
  \bigcup_{j=1}^{m} f_k(t_j) \cup \{r_k(v(t_1, \ldots, t_m))\} & \text{otherwise} 
\end{cases}
\tag{4.2}
\]

\[
s_k(v(t_1, \ldots, t_m)) = \bigcup_{j=1}^{m} s_k(t_j) \cup \begin{cases} 
  \emptyset & \text{if } \text{height}(v(t_1, \ldots, t_m)) > k \\
  v(t_1, \ldots, t_m) & \text{otherwise} 
\end{cases}
\tag{4.3}
\]
CHAPTER 4. IE BY TREE AUTOMATON INFERENCE

Note that the $k$-root and the $k$-subtrees have height at most $k$, and that the $k$-forks have height exactly $k$.

Following (García and Vidal 1990), we can define the following. A language $L$ is the class of $k$-testable tree languages in the strict sense ($k$-TTLSS) if there exist a $k$, a set of labels $V$, and sets $\mathcal{R}$, $\mathcal{F}$ and $\mathcal{S}$, such that $t \in L$ iff $t \in T_k$, $r_{k-1}(t) \subseteq \mathcal{R}$, $f_k(t) \subseteq \mathcal{F}$, and $s_{k-1}(t) \subseteq \mathcal{S}$.

These languages are thus characterized by the $(k-1)$-root trees that appear at the root (upper part of the tree), the set of $k$-fork trees that appear in the middle part, and the set of $(k-1)$-subtrees that appear in the lower part (at the leaves and nearby).

**Example 4.1** Suppose $t = a(b(a(b(x)), c)$ then $r_2(t) = \{a(b, c)\}; f_2(t) = \{a(b, c), b(a), a(b, x)\};$ and $s_2(t) = \{a(b, x), b, x, c\}.$

The level of a node is defined as the number of edges on the path from the node to the root. The skeleton of tree $t$, $\text{skeleton}(t)$, is defined as $t$ with all of its labels, except the root label, changed to a wildcard *.

For example, $\text{skeleton}(a(b(d), c(e, f))) = a(*, *, *, *)))$. A partition of a set $S$ is a set of disjoint nonempty subsets of $S$ (called classes) such that the union of the subsets is $S$. The children of a tree $\nu(t_1, \ldots, t_m)$ are $t_1, \ldots, t_m$. A tree $t$ covers a tree $t'$ if $t'$ can be derived from $t$ by replacing some of the wildcards in $t$.

### 4.5.5 The $k$-testable algorithm

The $k$-testable algorithm (García 1993; Knautila 1993; Rico-Juan, Calera-Rubio, and Carrasco 2000) is parameterized by a natural number $k$; its name comes from the notion of a “$k$-testable tree language”. Informally, a tree language (set of trees) is $k$-testable if membership of a tree in the language can be determined just by looking at its $(k - 1)$-root, $(k-1)$-forks, and $(k-1)$-subtrees. The $k$-testable algorithm is capable of identifying in the limit any $k$-testable tree language from positive examples only. We have selected it because the information to be extracted typically has a locally testable character. Intuitively, given an example, the right value of $k$ is the minimal value that ensures that the target $x$ and the distinguishing context are in the same fork.

The choice of $k$ is performed automatically using cross-validation, choosing the smallest $k$ giving the best results. Our cross-validation approach takes randomly one half of the dataset for training and uses the rest for testing. First calculating a score for $k = 2$, the value for $k$ is increased until the score shows a decrease. The least $k$-value giving a maximal score is then selected as the best value. As argued in Chapter 5, the F1-score we use first increases and then decreases, hence this is appropriate to choose $k$.

The procedure to learn the tree automaton (Rico-Juan, Calera-Rubio, and Carrasco 2000) is shown in Algorithm 1. The algorithm uses the $(k - 1)$-roots, $(k-1)$-forks and $(k-1)$-subtrees occurring in the examples to derive states
and transitions. Note that the algorithm uses trees as states of the inferred automaton: the \((k-1)\)-roots, the \((k-1)\)-subtrees and the \((k-1)\)-roots of the \(k\)-forks all become states. It is a simple way to ensure that the state associated with a node not only depends on the label, but also on the states of the children.

**Algorithm 1.** \(k\)-testable

**Input:** A set \(T\) of positive examples (ranked trees over \(V\)) and a positive integer \(k\).

**Output:** A tree automaton \((V, Q, \Delta, FS)\).

1. \(F := \bigcup \{f_k(t) \mid t \in T\}\)
2. \(S := \bigcup \{s_{k-1}(t) \mid t \in T\}\)
3. \(FS := \{r_{k-1}(t) \mid t \in T\}\)
4. \(Q := S \cup FS \cup \{r_{k-1}(f) \mid f \in F\}\)
5. \(\Delta := \{(v, t_1, \ldots, t_m) \rightarrow v(t_1, \ldots, t_m) \mid v(t_1, \ldots, t_m) \in S\}\)
6. \(\Delta := \Delta \cup \{(v, t_1, \ldots, t_m) \rightarrow r_{k-1}(v(t_1, \ldots, t_m)) \mid v(t_1, \ldots, t_m) \in F\}\)

**Example 4.2** Applying Algorithm 1 on the term of Example 4.1 for \(k = 3\), we obtain:

- \(F = f_3(t) = \{a(b(a), c), b(a(b, x))\}\) and \(S = s_2(t) = \{a(b, x), b, x, c\}\).
- \(FS = r_2(t) = \{a(b, c)\}\)
- \(Q = \{a(b, c), b(a), a(b, x), b, x, c\}\)
- transitions:
  - \(a(b, x) \in S: (a, b, x) \rightarrow a(b, x)\)
  - \(b \in S: (b) \rightarrow b\)
  - \(x \in S : (x) \rightarrow x\)
  - \(c \in S : (c) \rightarrow c\)
  - \(a(b(a), c) \in F: (a, b(a), c) \rightarrow a(b, c)\)
  - \(b(a(b, x)) \in F: (b, a(b, x)) \rightarrow b(a)\)

With more (and larger) examples, more transitions are created and generalisation occurs: also trees different from the given ones will be labeled with an accepting state (a state from \(FS\)).

### 4.5.6 The g-testable algorithm

The basic idea of the g-testable algorithm is to generalize the transitions originating from forks that are not important for the extraction. Important forks are those that contain the target label \(x\). In Algorithm 2, they are collected in the set \(TF\) (target forks), the other ones in the set \(OF\) (other forks). The
generalisation is parameterised by a level $l$. It replaces the label of a node by a wildcard when its level is greater or equal to $l$. The meaning of the wildcards is explained in the next paragraph. The algorithm uses a function $\text{gen}(f, l)$ for this generalisation. Figure 4.6 shows a fork $f$ (left), $\text{gen}(f, 1)$ (middle) and $\text{gen}(f, 2)$ (right). To prevent overgeneralisation, we require that the generalisation of a fork does not cover any target fork. The value of parameters $k$ and $l$ are determined by the same cross-validation method as explained above.

The meaning of a generalized fork is the set of all trees that can be obtained by instantiating labels for the wildcards. Generalized forks yield transitions with wildcards. For example, with $k = 3$ and $l = 2$, the fork $\text{gen}(f, 2)$ from the Figure would yield the transition $a(b, c(*, *)) \rightarrow a(b, c)$, which on a 5-label alphabet $V = \{a, b, c, d, e\}$ effectively stands for the $5^2 = 25$ possible transitions obtained by instantiating labels for the wildcards. Similarly, with $k = 3$ and $l = 1$, the fork $\text{gen}(f, 1)$ from the Figure would yield the transition $a(*, *(*, *)) \rightarrow a(*, *)$, which then stands for $5^4$ possible transitions obtained by instantiating labels for the wildcards on the left-hand side of the transition. Since the right-hand side stands for the 2-root of the fork, the wildcards on the right-hand side are instantiated in accordance with the left-hand side. Some concrete example instantiations of the transition are:

- $a(b, c(d, e)) \rightarrow a(b, c)$
- $a(d, c(b, c)) \rightarrow a(d, e)$
- $a(a, a(b, b)) \rightarrow a(a, a)$

The detailed $g$-testable procedure is shown in Algorithm 2. Note that the $(k - 1)$-subtrees of $k$-forks are explicitly added as states. In the $k$-testable algorithm, they were already present as $(k - 1)$-subtrees or as $(k - 1)$-roots of other forks. For generalized $k$-forks, this is no longer the case, hence their subtrees are explicitly added. Note also that with $l = k$, the $g$-testable algorithm will output exactly the same automaton as the $k$-testable algorithm.

**Example 4.3** Applying Algorithm 2 on the term of Example 4.1 for $k = 3$ and $l = 2$, we obtain:
4.5 IE WITH RANKED TREE ALGORITHMS

Algorithm 2 g-testable algorithm

\textbf{Input:} A set \( T \) of positive examples, parameters \( k \) and \( l \).

\textbf{Output:} A tree automaton \((V, Q, \Delta, FS)\)

\begin{enumerate}
  \item \( \mathcal{F}_0 := \bigcup \{ f_k(t) \mid t \in T \} \)
  \item \( \mathcal{T} = \{ f \in \mathcal{F}_0 \mid f \text{ contains } x \} \)
  \item \( \mathcal{O}_0 := \mathcal{F}_0 - \mathcal{T} \)
  \item \( \mathcal{O}_{\text{gen}} := \{ f \in \mathcal{O}_0 \mid \text{gen}(f, l) \text{ covers one of } \mathcal{T} \} \)
  \item \( \mathcal{O}_{\text{common}} := \mathcal{O}_0 - \mathcal{O}_{\text{gen}} \)
  \item \( \mathcal{F} := \mathcal{T} \cup \{ \text{gen}(f, l) \mid f \in \mathcal{O}_{\text{gen}} \} \cup \mathcal{O}_{\text{common}} \)
  \item \( s := \bigcup \{ s_k(t) \mid t \in T \} \)
  \item \( FS := \{ v(t_1, \ldots, t_m) \mid v(t_1, \ldots, t_m) \in S \} \)
  \item \( Q := s \cup FS \cup \{ r_k(f) \mid f \in \mathcal{F} \} \cup \bigcup \{ s_k(t) \mid t \in T \} \)
  \item \( \Delta := \{ (v(t_1, \ldots, t_m) \rightarrow v(t_1, \ldots, t_m)) \mid v(t_1, \ldots, t_m) \in S \} \)
  \item \( \Delta := \Delta \cup \{ (v(t_1, \ldots, t_m) \rightarrow r_k(v(t_1, \ldots, t_m))) \mid v(t_1, \ldots, t_m) \in \mathcal{F} \} \)
\end{enumerate}

\begin{itemize}
  \item \( \mathcal{F}_0 = f_0(t) = \{ a(b(a), c), b(a(b, x)) \} \)
  \item \( \mathcal{T} = \{ b(a(b, x)) \} \)
  \item \( \mathcal{O}_0 = \{ a(b(a), c) \} \)
  \item \( \mathcal{O}_{\text{gen}} = \emptyset \)
  \item \( \mathcal{O}_{\text{common}} = \{ a(b(a), c) \} \)
  \item \( \mathcal{F} = \{ a(b(*), c), b(a(b, x)) \} \)
  \item \( s_2(t) = \{ a(b(x), b, x, c) \} \)
  \item \( FS = r_2(t) = \{ a(b, c) \} \)
  \item \( Q = \{ a(b, c), b(a), a(b, x), b, x, c, b(*) \} \)
  \item \textbf{transitions:}
    \begin{itemize}
      \item \( a(b, x) \in S : (a, b, x) \rightarrow a(b, x) \)
      \item \( b \in S : (b) \rightarrow b \)
      \item \( x \in S : (x) \rightarrow x \)
      \item \( c \in S : (c) \rightarrow c \)
      \item \( a(b(*), c) \in \mathcal{F} : (a, b(*), c) \rightarrow a(b, c) \)
      \item \( b(a(b, x)) \in \mathcal{F} : (b, a(b, x)) \rightarrow b(a) \)
    \end{itemize}
\end{itemize}

\textbf{Example 4.4} Applying Algorithm 2 on the term of Example 4.1 for \( k = 3 \) and \( l = 1 \), we obtain:
\[ F_0 = f_3(t) = \{ a(b(a), c), b(a(b, x)) \} \]

\[ \mathcal{T}F = \{ b(a(b, x)) \} \]

\[ OF_0 = \{ a(b(a), c) \} \]

\[ OF_{nogen} = \emptyset \]

\[ OF_{gen} = \{ a(b(a), c) \} \]

\[ \mathcal{F} = \{ a(*(*)), b(a(b, x)) \} \]

\[ S = s_2(t) = \{ a(b, x), b, x, c \} \]

\[ FS = r_2(t) = \{ a(b, c) \} \]

\[ Q = \{ a(b, c), b(a, a(b, x), b, x, c, a(*, *), *(*), *) \} \]

transitions:

- \( a(b, x) \in S : (a, b, x) \rightarrow a(b, x) \)
- \( b \in S : (b) \rightarrow b \)
- \( x \in S : (x) \rightarrow x \)
- \( c \in S : (c) \rightarrow c \)
- \( a(*(*), *) \in \mathcal{F} : (a, *(*), *) \rightarrow a(*, *) \)
- \( b(a(b, x)) \in \mathcal{F} : (b, a(b, x)) \rightarrow b(a) \)

\[ \diamond \]

4.5.7 The \( gl \)-testable algorithm

As the \( k \)-testable algorithm, the \( gl \)-testable algorithm has a single parameter \( k \), whose optimal value is determined by the same cross-validation method. As the \( g \)-testable algorithm, the \( gl \)-testable algorithm divides forks in target forks \( \mathcal{T}F \) and other forks \( OF \) and generalises the other forks. However, the amount of generalisation is not determined by a second parameter \( l \), but by a more exhaustive exploration of possible generalisations.

To check that some generalisation is not overly general, we perform a cover test against the target forks (as in the \( g \)-testable algorithm) but also another test. This second test checks that the children of a fork (which are also states), are not covered by states containing the target \( x \). Such states can originate from subtrees of height \( k - 1 \). These states (target subtrees) are collected in the set \( \mathcal{T}S \).

To avoid an exhaustive search over all possible generalisations of a fork, some heuristics are used. As a first heuristic, the other forks are partitioned according to their skeleton by means of a procedure \( \text{partition} \) (not shown). For instance, with \( OF = \{ a(d), a(c, d), a(h), a(c), a(b, c), a(c, d, e) \} \), \( \text{partition}(OF) = \)
### 4.5 IE WITH RANKED TREE ALGORITHMS

\[
t : \quad a \quad a \quad a
\]

\[
\text{b} \quad \text{c} \quad \text{b} \quad \text{c} \\
\text{d} \quad \text{e} \quad \text{f} \quad \text{d} \quad * \quad \text{f} \\
\]

**Figure 4.7:** A fork and its two possible generalizations.

\{ \{a(b), a(c), a(d)\}, \{a(b, c), a(c, d)\}, \{a(c, d, e)\} \}. Then the procedure `pgen` (also not shown) computes a single generalisation of the forks in the same class (with the same skeleton): common labels are preserved but all other labels are replaced by a wildcard `*`. For instance, `pgen(\{a(b(c, d), a(b(c, e), a(b(f), e)) = a(b(\ast), \ast)`. If this generalisation is not overly general, it is used as initial value for a search of further generalisations. Otherwise, each fork of the partition is considered for generalisation. This further generalisation is performed by the procedure `gend` (Algorithm 4) with as inputs a set of target subtrees, a set of target forks (both used to check against overgeneralisation) and the set of forks to be generalised. This procedure returns the most general forks that are allowed by the overgeneralisation check. It only considers generalisations at the bottommost level (i.e., introducing wildcards only for leaves that are at depth \(n\), with \(n\) the height of the tree); this is a heuristic decision inspired by earlier experimental results.

To reduce redundancy, the partial order between generalisations is exploited. One position at a time is generalised and checked for overgeneralisation before considering further generalisations.\(^1\) The procedure `gentree` (Algorithm 5) returns all acceptable forks with one position generalised.

Consider for instance a fork \(a(b(d), c(e, f))\). It has seven possible generalisations for the labels at the bottommost level. They are partially ordered by the covers relation. The most specific ones are \(a(b(\ast), c(e, f))\), \(a(b(d), c(\ast, f))\) and \(a(b(d), c(e, \ast))\) (the ones tested by `gentree` when passed the initial fork), the more general ones are \(a(b(\ast), c(\ast, f)), a(b(\ast), c(e, \ast))\) and \(a(b(d), c(\ast, \ast))\) and \(a(b(\ast), c(\ast, \ast))\) is the most general one. The fork and two of its generalisations are shown in Figure 4.7.

The detailed gl-testable procedure is shown in Algorithm 3. The used heuristics were determined by experiment.

**Example 4.5** Applying Algorithm 3 on the term of Example 4.1 for \(k = 3\), we obtain:

\(^1\)A further optimisation would be to store unacceptable generalisations, and exploit the fact that any generalisations of these are automatically unacceptable as well. E.g., if \(a(b, c, \ast)\) is acceptable, then our algorithm considers its generalisation \(a(\ast, c, \ast)\); however, if it already knows that \(a(\ast, c, d)\) is unacceptable, then this new generalisation need not be considered. This further optimisation is not implemented in our algorithm.
Algorithm 3 gl-testable

Input: A set $T$ of positive examples and a positive integer $k$
Output: A tree automaton $(V, Q, \Delta, FS)$
1: $F_0 := \bigcup\{f_k(t) \mid t \in T\}$
2: $TF := \{f \in F_0 \mid f \text{ contains } x\}$
3: $OF := F_0 - TF$
4: $F = TF$
5: $P = \text{partition}(OF)$
6: $FS := \{r_{k-1}(t) \mid t \in T\}$
7: $S := \bigcup\{s_{k-1}(t) \mid t \in T\}$
8: $TS = \{s \in S \mid s \text{ contains } x, \text{ height}(s) = k - 1\}$
9: for each $C \in P$ do
10: \hspace{1em} $c = \text{pgen}(C)$ \hspace{1em} % candidate generalization
11: \hspace{1em} if $c$ covers one of $TF$ or one of $\text{children}(c)$ covers one of $TS$ then
12: \hspace{2em} $F = F \cup \text{genl}(TS, TF, C)$
13: \hspace{1em} else
14: \hspace{2em} $F = F \cup \text{genl}(TS, TF, \{c\})$
15: \hspace{1em} end if
16: end for
17: $Q := S \cup FS \cup \{r_{k-1}(f) \mid f \in F\} \cup \bigcup\{s_{k-1}(f) \mid f \in F\}$
18: $\Delta := \{(v, t_1, \ldots, t_m) \rightarrow v(t_1, \ldots, t_m) \mid v(t_1, \ldots, t_m) \in S\}$
19: $\Delta := \Delta \cup \{(v, t_1, \ldots, t_m) \rightarrow r_{k-1}(v(t_1, \ldots, t_m)) \mid v(t_1, \ldots, t_m) \in F\}$

Algorithm 4 genl

Input: Sets $TS$ of target subtrees, $TF$ of target forks, and $T$ of trees
Output: A set of trees $G$ (a generalisation of $T$)
1: $G := \emptyset$
2: while $T \neq \emptyset$ do
3: \hspace{1em} select $t$ from $T$ and remove it
4: \hspace{1em} $C := \text{gentree}(TS, TF, t)$
5: \hspace{1em} if $C = \emptyset$ then
6: \hspace{2em} $G := G \cup \{t\}$
7: \hspace{1em} else
8: \hspace{2em} $T := T - \{t \mid t \in T \text{ and } t \text{ is covered by some } c \in C\}$
9: \hspace{2em} $T := T \cup C$
10: \hspace{1em} end if
11: end while
4.6. IE WITH UNRANKED TREE ALGORITHM

Algorithm 5 genTree

Input: Sets $\mathcal{T S}$ of target subtrees, $\mathcal{TF}$ of target forks, and a tree $t$
Output: A set of trees $C$

1. $C_0 := \{t' \mid t'$ is derived from $t$ by replacing one bottommost label $\neq *$ by $*$\}
2. $C := C_0 - \{c \in C_0 \mid c$ covers one of $\mathcal{TF}$, or one of children $(c)$ covers one of $\mathcal{TS}\}$

- $FS = r_2(t) = \{a(b, c)\}$, $QF = \{a(b(a), c)\}$, $\mathcal{TF} = \{b(a(b), x)\}$ and $S = s_2(t) = \{a(b, x), b, x, c\}$.
- $\mathcal{TS} = \{a(b, x)\}$
- $\mathcal{P} = \{\{a(b(a), c)\}\}$
- $\mathcal{F} = \{a(b(*), c), b(a(b, x))\}$
- $Q = \{a(b, c), b(a), b(*), a(b, x), b, x, c\}$
- transitions:
  - $a(b, x) \in S: (a, b, x) \rightarrow a(b, x)$
  - $b \in S: (b) \rightarrow b$
  - $x \in S: (x) \rightarrow x$
  - $c \in S: (c) \rightarrow c$
  - $a(b(*), c) \in F: (a, b(*), c) \rightarrow a(b, c)$
  - $b(a(b, x)) \in F: (b, a(b, x)) \rightarrow b(a)$

Note again that the use of wildcards in the representation of the sets $\mathcal{F}$, $Q$ and $\Delta$ in the example is really just an abbreviation; e.g., when $Q$ contains $b(*)$, this really means it contains the states $b(a), b(b), b(c)$ and $b(x)$.

4.6 IE with unranked tree algorithm

4.6.1 Introduction

In the previous section (Section 4.5), we have described three tree automaton inference algorithms, which are developed for wrapping HTML/XML documents, that expect ranked trees. Indeed, a simple way to apply these algorithms to HTML or XML documents, which are unranked trees, is to transform the latter into binary trees. The work in the previous section employs a $k$-testable tree
automaton as wrapper, which is based on (Garcia 1993; Knuntila 1993; Rico-Juan, Calera-Rubio, and Carasco 2000), and its two extensions: the g-testable algorithm and the g-testable algorithm.

Typically in IE task from structured documents, there is some structural context close to the target field. After linearization in a string, this context can be arbitrarily far away, which makes the learning task very difficult for string based methods. While binarisation may also increase the distance between the context and the target field, they remain closer. Thus, the learning task should be easier in this situation. This is confirmed by the experiments in Chapter 5. If distance between the relevant context and the target is indeed the main factor determining the ability to learn an appropriate automaton, then an algorithm inducing a wrapper directly from the unranked tree should perform even better than algorithms that expect ranked trees.

This section develops a novel wrapper induction method that works directly on the unranked tree structure of the documents, which avoids binarization. As in the work of the previous section, user intervention is limited to annotating the field to be extracted in several representative examples. The use of unranked trees is so far a less explored alternative. Unranked tree automata formalisms exist, e.g., (Pair and Quere 1968; Takahashi 1975) and have been studied since the late 60's, see e.g. (Brüggenmann-Klein, Murata, and Wood 2001; Neven 2002) for a survey. They have transition rules of the form \( (v, e) \to q \), where \( e \) is a regular expression that describes a sequence of states. However to the best of our knowledge, algorithms for inducing them do not yet exist. The work in this section, which is introduced in (Kosala, Bruynooghe, Van den Bussche, and Blockeel 2003), is a first step in this direction.

4.6.2 Training and testing process

The training and testing procedures for unranked trees are basically equal to the training and testing procedures described for the ranked trees in Section 4.5.2, but without the conversion to binary trees. The learning procedure is as follows (see also Figure 4.1):

1. Replace in the examples the target field by 'x', the distinguishing context(s) (if present) by 'ctx' and all other text fields by CDATA.

2. Map examples to trees and run an unranked tree automaton inference algorithm on the examples and return the inferred automaton.

The extraction procedure is as follows:

1. Map the test document to a tree and replace the distinguishing context(s) (if present) by 'ctx' and all other text fields by CDATA.
2. Repeat for all CDATA nodes:
   - Replace the label of one CDATA node by the special label ‘x’.
   - Run the inferred tree automaton.
   - If the tree is accepted by the automaton, then output the original text of the node labeled with ‘x’.

The automaton can succeed for zero, one or more text nodes. The text nodes for which it succeeds are the extracted fields.

4.6.3 Definitions
Some definitions used in this section are based on definitions described in Section 4.5.4. Here we only give additional definitions.

Two different states (non-terminals) A1 and A2 are competing with each other if A1 and A2 share the same node label in their transitions. For example, \( (a(b)) \rightarrow A1 \) and \( (a(c)) \rightarrow A2 \), where \( b, c \in Q \), are competing with each other.

The class of local tree languages is a subclass of the \( k \)-TTLSS with \( k = 2 \). The class of 2-TTLSS is interesting because it could be used directly to describe the content model of HTML/XML documents. In fact, the class of local regular tree grammars roughly corresponds to DTD (Murata, Lee, and Mani 2001). Following (Murata, Lee, and Mani 2001), we can state the following: a local tree automaton is a tree automaton that does not have competing states.

4.6.4 Local unranked tree automaton inference algorithm
We propose offline and online algorithms to infer local unranked tree automata. We will first describe the offline version of the algorithm before describing the online version.

The offline procedure we designed to learn the local unranked tree automaton is shown in Algorithm 6. This algorithm takes as input a set of trees \( T \) as positive examples and a positive integer \( k, > 1 \), which is the parameter for the \( k \)-contextual subroutine that it calls. Algorithm 6 consists of two for-loops.

In the first for-loop, our algorithm collects all 2-forks, 1-subtrees and 1-roots. The latter become final states. However, before these steps, the node labels of each input tree \( t \) are rewritten using the function \( \text{convert}Labels(t) \). This function (shown in Algorithm 7) rewrites the node labels if they contain the special label ‘x’ or any distinguishing context. This is done recursively by propagating this information towards the root of the tree. An example of how this function works is shown in the upper part of Figure 4.8. The purpose of this function, which is shown in the lower part of Figure 4.8, is to make the special label and the context information remembered up to the root. This function actually makes the resulting tree automaton not purely local.
Next, the states are collected (the 1-root, the 1-subtrees, and the 1-roots of the forks) in $Q$, the transitions are initialized with one transition for each 1-subtree and the 2-forks are partitioned according to the label of the forks’ root. The latter results in a set of pairs $(v, Str)$ with $Str$ a set of sequences, each sequence being the children of a fork. E.g., $\{a, \{(b,c),(b,c,c)\}\}$ represents two forks with root label $a$.

In the second for-loop, the algorithm gives the sequences of the children of the fork trees in each partition, that have been collected in the first for-loop, as input to the $k$-contextual algorithm (Algorithm 8) that learns a deterministic finite automaton (DFA). This DFA can be used as the representation of the regular expressions $e$ to be used in the transitions $v(e) \rightarrow q$ of the UTA. Example 4.6 serves as an illustration of the $k$-contextual algorithm.

**Example 4.6** Consider an input string $ab$ for $k = 3$. The value of $U$ is $\#\#ab\#$ and one obtains $FS = \{b\#$ and $\Delta = \{\#\#a \rightarrow \#a, \#ab \rightarrow ab, \#ab \rightarrow b\#$}, where $\#$ is a distinguished label that is not in $\Sigma$.

The $k$-contextual algorithm (Muggleton 1990; Ahonen 1996; Garcia and Vidal 1990) is an algorithm that is capable of identifying in the limit any $k$-contextual string automaton, a subset of finite automata, from positive examples only. In principle, we could use any string automaton inference al-
4.6 IE WITH UNRANKED TREE ALGORITHM

algorithm for this purpose. There have been many algorithms developed for inferring finite automata, for example see e.g. (Murphy 1996; Parekh and Honavar 1998; Salsakibara 1997). We choose the $k$-contextual algorithm because it is efficient, simple, and works well in practice. Moreover it could learn an automaton from just one example, which is useful for learning from partitions that contain one fork only. In fact, this algorithm is the string counterpart of the $k$-testable tree automaton inference algorithms proposed in (Garcia 1993; Knuntila 1993).

An element $(v, S_{tr})$ of the partition gives all positive examples for a particular label $v$. The regular expression $e$ captures the regularity in these examples (the document content model in XML terminology). To obtain sufficient generalization, we decided, after some experimentation, to distinguish three cases. If all children of a $v_e$ node are long enough, we construct a DFA using the $k$-contextual algorithm with $k$-value $k_e$, otherwise with value 2. For other labels (either a $v_{ctx}$ label or an original label), we ignore the content of the children and accept any sequence (the regular expression '*').

The reason behind this is as follows. In order to allow correct identification of the place of the field that needs to be extracted, sometimes a large value of $k_e$ is needed. Using one large value of $k_e$ for all the nodes' children makes the resulting automaton too specific. Therefore we require the algorithm to apply the large $k_e$ value only to the nodes' children that contain the special label 'x'. This constraint, however, is still not enough to restrain the resulting automaton from being too specific in our experiments. Therefore, similar to the idea used in the $g$-testable and $gl$-testable algorithms, we generalize the nodes' children when 'x' does not belong to them because these transitions are not important for the extraction. We generalize the nodes' children and keep the node labels intact. In this way the acceptance of the children is universal but the resulting state is not.

Note that we denote the state of a tree $(\delta(v(e)) = r_1(v(e)))$ with its label $v$ to simplify the notation and inference procedure because local languages have no competing states.

It is quite straightforward to make an incremental or online version of Algorithm 6. This is based on the result of (Muggleton 1990), which states that the $k$-contextual algorithm is incremental when $k_e$ is fixed in advance. Because the $e$ is represented by a $k$-contextual automaton, the updating of the automaton can be done by just adding the new $k$-grams with their states to the previous automaton (Ahonen 1996). The incremental or online version of the Algorithm 6 is shown in Algorithm 9.

Angluin (Angluin 1980) gives a characterization of the classes of languages that are identifiable in the limit from positive examples only. A sufficient condition is established by the following theorem:
Algorithm 6 Local unranked tree inference algorithm (offline)
Input: A set $T$ of positive examples and a positive integer $k_c > 1$.
Output: An unranked tree automaton $(V, Q, \Delta, FS)$.
1: $FS = F = S = \emptyset$;
2: for each $t \in T$ do
3: $t' = \text{convertLabels}(t)$
4: $F = F \cup f_2(t')$ % 2-forces
5: $S = S \cup s_1(t')$ % 1-subtrees
6: $FS = FS \cup r_1(t')$ % 1-root
7: end for
8: $Q = FS \cup \{r_1(f)|f \in F\} \cup S$
9: $\Delta = \{v(e) \rightarrow v|v \in S\}$
10: $P = \text{partition}(F)$
11: for each $(v, Str) \in P$ do
12: if $v = v_2$ and the length of the shortest string in $Str \geq k_c$ then
13: $e = k_{\text{contextual}}(k_c, Str)$ % a DFA
14: else if $v = v_2$ then
15: $e = k_{\text{contextual}}(2, Str)$ % a DFA
16: else
17: $e = \ast$ % accepts any sequence
18: end if
19: $\Delta = \Delta \cup \{v(e) \rightarrow v\}$
20: end for

Algorithm 7 \text{convertLabels}(t)
Input: A tree $t = v(t_1, \ldots, t_n)$
Output: A tree $t'$
1: $t'_1 = \text{convertLabels}(t_1), \ldots, t'_n = \text{convertLabels}(t_n)$
2: if $\exists : t'_i$ has label $l_c$ or $x$ then
3: $t' = v_2(t'_1, \ldots, t'_n)$
4: else if $\exists : t'_i$ has label $l_{ctx}$ or $ctx$ then
5: $t' = v_{ctx}(t'_1, \ldots, t'_n)$
6: else
7: $t' = t$
8: end if
Algorithm 8 $k_{\text{contextual}}(k, \text{Str})$

Input: A set of strings $\text{Str}$ over $\Sigma$ and a positive integer $k > 1$.

Output: A $k$-contextual DFA $(\Sigma, Q, \Delta, FS)$.

1. $Q = FS = \Delta = \emptyset$;
2. $U = \{\#^{k-1}s|s \in \text{Str}, \# \notin \Sigma\}$
3. $\text{Grams} = \{s|s \text{ is a substring of } u, u \in U, |s| = k\}$
4. for each $g \in \text{Grams}$ do
   5. Let $g = uv$ with $|u| = 1$ and $|v| = k - 1$
   6. $Q = Q \cup \{v\}$
   7. if $v = v'\#$ then
      8. $FS = FS \cup \{v\}$
   9. end if
10. $\Delta = \Delta \cup \{uv \rightarrow v\}$
11. end for

Algorithm 9 Local unranked tree inference algorithm (online)

Input: A set $T$ of positive examples and a positive integer $k_c > 1$.

Output: An unranked tree automaton $(V, Q, \Delta, FS)$.

1. $FS = F = S = \emptyset$;
2. for each $t \in T$ do
   3. $t' = \text{convert}\_\text{labels}(t)$
   4. $F = F \cup f_2(t')$ % 2-fooks
   5. $S = S \cup s_1(t')$ % 1-subtrees
   6. $FS = FS \cup r_1(t')$ % 1-root
   7. $P = \text{partition}(F)$
   8. for each $(v, \text{Str}) \in P$ do
      9. if $v = v_x$ and $|\text{Str}| \geq k_c$ then
         10. $e = k_{\text{contextual}}(k_c, \text{Str})$ % a DFA
      11. else if $v = v_x$ then
         12. $e = k_{\text{contextual}}(2, \text{Str})$ % a DFA
      13. else
         14. $e = *$ % accepts any sequence
      15. end if
      16. $\Delta = \Delta \cup \{v(e) \rightarrow v\}$
   17. end for
18. end for

19. $Q = FS \cup \{r_1(f)|f \in F\} \cup S$
20. $\Delta = \Delta \cup \{v(e) \rightarrow v|v \in S\}$
Theorem 4.1 (Angluin, 1980) Let \( L_1, L_2, \ldots \) be an indexed class of recursive languages such that, for every nonempty finite set \( T \subseteq \Sigma^* \), the cardinality of the set \( C(T) = \{ L : T \subseteq L \text{ and } L = L_i \text{ for some } i \} \) is finite. Then this class is identifiable in the limit from only positive examples.

Following (Garcia and Vidal 1990), it can be shown that because the number of four-tuples \((V, \mathcal{R}, \mathcal{F}, \mathcal{S})\) such that \( V = V(T) \) (the alphabet from the set of positive examples \( T \)), \( r_{k-1}(T) \subseteq \mathcal{R}, f_k(T) \subseteq \mathcal{F}, s_{k-1}(T) \subseteq \mathcal{S} \), is finite, the set of different \( k \)-testable tree languages in the strict sense (\( k \)-TTLSS) containing \( T \) is finite. From Theorem 4.1, this implies that the class of \( k \)-TTLSS is identifiable in the limit using positive examples only and we can prove the following:

**Theorem 4.2** Every unranked tree language that is definable by a local unranked tree automaton with \( k \)-contextual regular expressions is identifiable in the limit, from positive examples only, by our algorithm.

**Example 4.7** Applying Algorithm 6 on the tree of Example 4.1 for \( k_c = 2 \), we obtain:

- \( t' = a_x(b_x(a_x(b, x)), c) \)
- \( r_1(t') = \{ a_x \} \)
- \( \mathcal{F} = f_2(t') = \{ a_x(b_x,c), b_x(a_x), a_x(b, x) \} \)
- \( \mathcal{S} = s_1(t') = \{ b, x, c \} \)
- \( FS = \{ a_x \} \)
- \( Q = \{ a_x, b_x, b, x, c \} \)
- \( P_1 = \{ a_x, \{(b_x,c)\} \} \)
- \( P_2 = \{ b_x, \{a_x\} \} \)
- \( \text{DFA}_1: \Delta = \{ \# b_x \rightarrow b_x; b_x c \rightarrow c; c \# \rightarrow \#; \# b \rightarrow b; b x \rightarrow x; x \# \rightarrow \# \}, Q = \{ b_x, c, \#, b, x \}, FS = \{ \# \} \).
- \( \text{DFA}_2: \Delta = \{ \# a_x \rightarrow a_x; a_x \# \rightarrow \# \}, Q = \{ a_x, \# \}, FS = \{ \# \} \).

**transitions:**
- \( b \in \mathcal{S} : b(e) \rightarrow b \)
- \( x \in \mathcal{S} : x(e) \rightarrow x \)
- \( c \in \mathcal{S} : c(e) \rightarrow c \)
- \( P_1 : a_x(\text{DFA}_1) \rightarrow a_x \)
- \( P_2 : b_x(\text{DFA}_2) \rightarrow b_x \)

\(\square\)
4.7 Summary

In this chapter, we have discussed in detail our motivation for tree-based IE methods. We have discussed how tree automata based methods differ from other tree-based approaches. Then, we have shown how the IE problem can be solved by grammatical inference. After that, we have presented two practical issues involved and several characteristics of our tree-based method in detail.

In the next part of this chapter, we have proposed several algorithms for tree-based IE. We have shown, in detail, the steps involved in tree-based IE with ranked tree algorithms: the $k$-testable tree automaton inference algorithm, and its two generalizations: the $g$-testable algorithm and the $gl$-testable algorithm. These generalizations have been proposed to deal with the problem of binarization of unranked trees.

We have also explored the second alternative to tree-based IE. This less explored alternative to tree-based IE infers an unranked tree automaton directly, instead of relying on binarization. Specifically, we have proposed an algorithm to infer local unranked tree automata and have shown that it converges in the limit.

We will describe the experiments on several IE tasks in the next chapter, using the tree automaton inference algorithms proposed in this chapter.
Chapter 5

IE Experiments

5.1 Introduction

In this chapter, we test and experiment with the tree automaton inference algorithms described in the previous chapter. This chapter basically consists of two parts, which are as follows.

In the first part, we describe experiments with three ranked tree automaton inference algorithms: the \( k \)-testable tree automaton inference algorithm, and its two generalizations: the \( g \)-testable algorithm and the \( gl \)-testable algorithm. In this part, we perform experiments in two different ways:

1. Experiments on some benchmark datasets are performed to obtain a close comparison with state-of-the-art string-based learners. Then a more detailed description of the experiments is given to investigate the characteristics of the family of \( k \)-testable languages, and

2. Experiments that focus on the \( gl \)-testable algorithm only are performed with larger datasets to measure its performance.

In the second part, we describe experiments with the local unranked tree automaton inference algorithm. Here, we also perform experiments in two different ways:

1. Experiments on the same benchmark datasets are performed to obtain a close comparison with the previous approaches, and

2. Experiments that focus on the efficiency of the unranked algorithm are performed with larger datasets and the results are compared with those from an analysis of the algorithm based on the PAC framework.

101
We also describe additional experiments on a proprietary syllabus dataset, which show a limitation of our tree-based methods. Finally, we discuss the advantages and limitations of our tree-based methods.

5.2 Experiments with the ranked tree automaton inference algorithms

The ranked tree automaton inference algorithms, which are used in this section, are described in Section 4.5 of Chapter 4.

These three ranked algorithms, $k$-testable, $g$-testable and $gl$-testable algorithms, are evaluated in two different ways:

- In the first experiment, we evaluate our method on several difficult benchmark data sets that are commonly used in IE research. We use the same setting as used by some previous work on wrapper induction in order to obtain a close comparison.

- In the second experiment, we evaluate our method in two relatively large, but simpler datasets, to evaluate its performance.

5.2.1 Test on the benchmark datasets

5.2.1.1 The datasets

We evaluated our method on some semi-structured data sets commonly used in IE research:\footnote{Available from http://www.isi.edu/~muslea/RISE/}

- The Internet Address Finder (IAF) dataset: a collection of web pages containing people's contact addresses, and

- The Quote Server (QS) dataset: a collection of web pages about stock quotes.

Within these two datasets, we have four extraction tasks to be performed. The tasks are the extraction of alternative and organization fields in the IAF dataset and the date and volume fields in the QS dataset. There are 10 example documents in each of these datasets. The number of fields to be extracted is respectively 94 (IAF-organization), 12 (IAF-alt.name), 24 (QS-date), and 25 (QS-vol). The motivations to choose these datasets are as follows.

1. Firstly, they are benchmark datasets that are commonly used for research in information extraction, so we can compare the results of our method directly with the results of other methods.
5.2 EXPERIMENTS WITH RANKED TREE ALGORITHMS

2 Secondly, they are the most difficult benchmarks we are aware of that require the extraction of a whole node of the document tree (our methods are designed for that task). In fact, one of the authors in (Muslea, Minton, and Knoblock 1999) has tried to build a handcrafted extractor given all available documents from the QS dataset and achieved only 88% accuracy.

We also test the k-testable, g-testable, and gl-testable algorithms on a small and simplified Shakespeare\(^2\) data set. This dataset is a significantly reduced version of Jon Bosak's Shakespeare XML dataset, which can be found from 
http://www.ibiblio.org/bosak/; its its DTD is shown in Figure 5.1.

In this dataset, an act consists of a title followed by one or more scenes; a scene consists of a title followed by one or more speech acts; a speech act consists of a speaker followed by one or more lines; finally title, speaker and line are normal strings of text. We use it to test the expressiveness of our methods. For this dataset, the task is to extract the title of the second scene of every act, in a particular play. The motivation to test on this data set is that we believe that the extraction task is very difficult for string-based methods even on the simplified data. This is because each scene has a complex structure of varying length. We used the simplified Shakespeare dataset because our three algorithms above performed very poorly on the full dataset, which DTD is shown in Figure 5.6. The latter is a consequence of the conversion to ranked trees: an 'act' field, for instance, can have many children, many of which precede the second scene of the act. After the conversion, this second scene then ends up very deep in the subtree below the 'act' tag, making it very difficult to identify using a k-testable automaton. Thus, we expect the simplified data set to be a good example of a task that is difficult for string-based wrappers but manageable for tree-based ones.

5.2.1.2 Evaluation criteria

The training and the testing processes follow the procedures outlined in Section 4.5 of Chapter 4. For evaluating our method, we use criteria that are

commonly used in the information retrieval research community:

- Precision $P$ is the number of correctly extracted objects divided by the total number of extractions,
- Recall $R$ is the number of correct extractions divided by the total number of objects present in the answer template, and
- F1 score is defined as $2P.R/(P + R)$, the harmonic mean of $P$ and $R$.

Alternatively, precision $P$ and recall $R$ can be defined as follows.

- $P = TP/(TP + FP)$
- $R = TP/(TP + FN)$

Where, $TP$ is true positive (the number of correct extraction), $FP$ is false positive (the number of incorrect extraction), $FN$ is false negative (the number of correct objects (or target fields) that were not extracted), $TP + FP$ is the total number of extraction, and $TP + FN$ is the total number of objects (or target fields) present in the answer template.

The precision, recall and F1 scores are calculated for each extracted slot (or field), without collecting different scores in a case frame.

### 5.2.1.3 Summary of the results

Table 5.1 shows the results we obtained as well as those obtained by some current state-of-the-art string-based methods: an algorithm based on Hidden Markov Models (HMMs) (Freitag and McCallum 1999), the Stalker wrapper induction algorithm (Muslea, Minton, and Knoblock 2001) and BWI (Freitag and Kushmerick 2000). We give results for the k-testable algorithm (as we reported in (Kosala, Van den Bussche, Bruynooghe, and Blockeel 2002)), the g-testable algorithm (as we reported in (Kosala, Bruynooghe, Blockeel, and Van den Bussche 2002)), and the gl-testable algorithm. The results for HMM, Stalker and BWI are adopted from (Freitag and Kushmerick 2000). All tests are performed with 10-fold cross validation following the splits used in (Freitag and Kushmerick 2000), except in the small Shakespeare dataset where 2-fold cross validation was used. Each split has 5 documents for training and 5 for testing. We refer to Chapter 6 for a brief description of the other methods.

Table 5.1 shows the results of the k-testable, g-testable and gl-testable algorithms for the optimal $k$ value (more specifically, $k$ was optimised during the first fold of the cross-validation, this optimal value was then used for all the other folds). As can be seen, our methods perform better in most of the test cases than the existing state-of-the-art string-based methods. The only exception is the field date in the QS dataset where BWI performs better. Compared
5.2 EXPERIMENTS WITH RANKED TREE ALGORITHMS

<table>
<thead>
<tr>
<th></th>
<th>IAF-all.name</th>
<th>IAF-organization</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Prec</td>
<td>Rec</td>
</tr>
<tr>
<td>HMM</td>
<td>1.7</td>
<td>90</td>
</tr>
<tr>
<td>Stalker</td>
<td>100</td>
<td>-</td>
</tr>
<tr>
<td>BWI</td>
<td>90.9</td>
<td>43.5</td>
</tr>
<tr>
<td>k-testable</td>
<td>100</td>
<td>73.9</td>
</tr>
<tr>
<td>g-testable</td>
<td>100</td>
<td>73.9</td>
</tr>
<tr>
<td>gl-testable</td>
<td>100</td>
<td>84.8</td>
</tr>
</tbody>
</table>

QS-date | QS-volume | Small Shakespeare
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Prec</td>
<td>Rec</td>
<td>F1</td>
</tr>
<tr>
<td>HMM</td>
<td>36.3</td>
<td>100</td>
</tr>
<tr>
<td>Stalker</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>BWI</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>k-testable</td>
<td>100</td>
<td>60.5</td>
</tr>
<tr>
<td>g-testable</td>
<td>100</td>
<td>60.5</td>
</tr>
<tr>
<td>gl-testable</td>
<td>100</td>
<td>60.5</td>
</tr>
</tbody>
</table>

Table 5.1: Comparison of the results

<table>
<thead>
<tr>
<th></th>
<th>IAF-all.name</th>
<th>IAF-org</th>
<th>QS-date</th>
<th>QS-volume</th>
<th>Shakespeare</th>
</tr>
</thead>
<tbody>
<tr>
<td>k-testable (k)</td>
<td>4</td>
<td>4</td>
<td>2</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>g-testable (k, l)</td>
<td>(5, 2)</td>
<td>(5, 2)</td>
<td>(3, 2)</td>
<td>(6, 5)</td>
<td>(4, 2)</td>
</tr>
<tr>
<td>gl-testable (k)</td>
<td>4</td>
<td>4</td>
<td>2</td>
<td>6</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 5.2: Parameters used for the experiments

to the results of k-testable, the gl-testable algorithm performs better in the IAF-alt.name, IAF-organization and small Shakespeare data. Compared to the results of g-testable, the gl-testable performs better in the IAF-alt.name and IAF-organization data but worse in the small Shakespeare data. We shall discuss these results in more detail below.

Table 5.2 shows the parameters k, (k, l) and k that were used by the k-testable, g-testable, and gl-testable algorithms respectively to produce the results in Table 5.1. A distinguishing context was used in the datasets IAF-alt.name and IAF-organization.

5.2.1.4 More detailed discussion of the results

In these datasets some of the best results with the gl-testable algorithm (i.e. in QS-volume and small Shakespeare data) are obtained with a k value bigger than the value used in the k-testable algorithm. This means that our goal, performing more generalisation while using bigger contexts, is achieved. Some other best results (i.e., in IAF-alt.name, IAF-organization and QS-date data)
are obtained with using the same \( k \) value. These results indicate that: (1) The wildcards are useful for our IE tasks as they can improve the results of the \( k \)-testable algorithm. (2) The two step generalisation, done by the procedure \textit{pgen} that generalises all forks in each partition and by the procedure \textit{gerl} that searches the generalisation of the bottommost labels more thoroughly, is useful for our IE tasks. This is shown by the better results of the \( gl \)-testable algorithm compared to the results of the \( g \)-testable algorithm in the two IAF datasets that are obtained with a smaller value of \( k \).

Despite the improvements of both \( gl \)-testable and \( g \)-testable algorithms in IAF-alt.name, IAF-organization and small Shakespeare datasets, they were not able to improve the results of the \( k \)-testable algorithm in the two QS datasets. For the QS-volume data, the reason is not clear to us. One explanation is that the result might be optimal for these learners given a certain set of training examples. For the QS-date data, the reason is that the inferred automaton is not general enough, as can be seen in Figure 5.2. In that figure, the recall of the most general automaton inferred (\( k = 2 \)) is not very high and the maximum precision is already reached with \( k = 2 \). Thus we cannot improve the F1 score by increasing the \( k \).

The \( gl \)-testable algorithm performs worse than the \( g \)-testable algorithm in the small Shakespeare data, although better in IAF-alt.name and IAF-organization datasets. The reason is that, besides optimising the \( k \) parameter, we also optimise the generalisation level of the \( g \)-testable algorithm by cross validation to suit a specific dataset. In other words, the \( g \)-testable algorithm has an extra parameter to optimise and is more heuristic in nature than the \( gl \)-testable algorithm.

Figure 5.2 shows how the F1 score of the \( gl \)-testable algorithm changes with \( k \). The solid line is the F1 score, while the other two dotted lines are precision (lightly dotted) and recall (heavily dotted). In this figure, we can clearly see the trade off between precision and recall. Actually, the behavior of the three tree-based algorithms that we test is quite similar. With small value of \( k \), the precision of these tree-based methods tend to be low because the automaton inferred is relatively general. As the value of \( k \) increases, the precision rises until a certain value of \( k \) where the maximum precision is reached. The recall of these tree-based methods behaves the other way around. For a low value of \( k \), the recall of these tree-based methods tend to be high. However as the value of \( k \) becomes higher, the recall decreases gradually. As the harmonic mean of the precision and recall, typically the F1 score curve starts with a low value at \( k = 2 \), increases, reaching a maximum and then starts to decrease. The decrease in the F1 score can be seen as a sign of overfitting. We use cross-validation based on the F1 score to optimize the parameters in Table 5.2 as a way to deal with overfitting.

Figure 5.3 shows the average training time of the \( k \)-testable and the \( gl \)-
5.2 EXPERIMENTS WITH RANKED TREE ALGORITHMS

Figure 5.2: The graphs of F1 score vs k

testable algorithms for different values of k. Overall, both algorithms show somewhat similar training times on our datasets. The theoretical running time of the k-testable algorithm is approximately \(O(kn)\), where \(n\) is the total size (or the total number of nodes) of the examples. The reasoning behind this is as follows. Using a suitable data structure for looking up forks and subtrees, finding a fork or subtree can be done in constant time. If \(k = 2\) then there are approximately \(n\) states in the trees. For \(k > 2\), more time is needed to cut the trees, to derive the forks and subtrees and there are more states. The latter can be approximated to be \(kn\). The derivation of the state transitions requires approximately \(O(n)\). Thus, the total running time of the \(k\)-testable is approximately \(O(kn)\). A similar upper bound holds for the time complexity of the \(g\)-testable algorithm.

The theoretical running time of the \(gl\)-testable algorithm is in the worst case exponential in the size of the subtrees, due to the finer search in the generalisation lattice. In our experiments, the \(gl\)-testable algorithm is still feasible to run if the \(k\) value used is less than 8. The preprocessing consists of parsing, conversion to the binary tree representation (both processes take time linear in the size of the document) and the manual insertion of the label \(x\). Our prototype implementation was implemented in Prolog and tested on a Pentium 1.7 GHz PC. Figure 5.3 shows that the actual training time (after preprocessing)
needed to infer the automaton is more or less linear in \( k \). One exception is the training time of the \( g \)-testable algorithm for IAF-alt.name which looks non-linear. The reason is that with \( k < 6 \) most candidate generalizations (the result of \( pgen \) function) do not suffer from overgeneralisation. Thus only one candidate generalization is input to the \( genl \) function. This is also the reason why in this task the \( g \)-testable algorithm is slightly faster than the \( k \)-testable algorithm for \( k < 6 \). At \( k = 6 \) several candidate generalizations suffer from overgeneralisation.

Actually the theoretical training time of the \( k \)-testable and the \( g \)-testable algorithms is better than that of BWI (Freitag and Kushmerick 2000), one of the string-based methods that are used for comparison. The training time of BWI increases exponentially with the increase of the look-ahead parameter. As reported in (Freitag and Kushmerick 2000) the IAF-alt.name, IAF-organization and QS-volume datasets need a long lookahead. They used lookaheads of 8 because these tasks need very long boundary detectors. We cannot compare the actual training time of BWI to ours in these datasets as it was not reported.

Figure 5.4 shows the average extraction time per document of the \( g \)-testable algorithms for different values of \( k \). The theoretical time complexity of the extraction procedure is \( O(n^2) \) where \( n \) is the number of nodes in the document. Indeed, the time of a single run is linear in the number of nodes (using suitable
data structures), while the automaton has to run for each replacement of a node by the target symbol $x$. Just for comparison, the extraction time of the tree automaton inferred by the $k$-testable algorithm (not shown here) is about two times faster than the extraction time of the generalised automaton inferred by the $gl$-testable algorithm. The reason is that the latter automaton needs additional time to match the wildcards.

Figure 5.5 shows the number of states, which is an indicator of the size of the automaton, inferred by the $k$-testable and $gl$-testable algorithms for different values of $k$. As we can see from the figure, the number of states inferred by the $gl$-testable algorithm is always smaller than the number of states inferred by the $k$-testable algorithm for $k > 2$. For $k = 2$, the number of states inferred is equal as the $gl$-testable algorithm performs no generalisation in this case.

### 5.2.2 Test on larger datasets

#### 5.2.2.1 The datasets

To evaluate the performance of our approach further, we test the $gl$-testable algorithm also on larger datasets. For the experiments below we use the Bigbook and the Okra datasets that are also available online\(^3\). In the Bigbook dataset we train the automaton to extract the 'name' and 'address' fields, and in the Okra dataset we train the automaton to extract the 'name' and 'email' fields. The Bigbook and Okra datasets contains 235 and 252 files respectively.

\(^3\)Available from http://www.isi.edu/~masiea/RISE/
The number of the name and address fields to be extracted from the Bigbook dataset is 4299 while the number of the name and email fields in the Okra datasets is 3334.

5.2.2.2 Experiment setting and the results

All experiments in this section are done with 10-fold cross-validation and the experimental setting is as follows. We divide both datasets in two parts.

The first part, consisting of 10 files (or documents), is used to test the generalization ability of the gl-testable algorithm. From these 10 files we obtain 184 and 120 examples from the Bigbook and Okra datasets respectively. First we determine for each extraction task the optimal $k$ by cross-validation in one random fold of training and test set. The best $k$ found is 5 for the tasks in the Bigbook dataset and 4 for the tasks in the Okra dataset. Then, from the same set of 184 (120) examples, we take randomly ten examples, then twenty examples, then thirty, and so on, and give them as training examples while the rest is for testing. This process is performed ten times for every extraction task. We stop when the induced automaton has an average F1 score of 98% or better on the test examples. The fourth column of Table 5.3 shows the number of examples needed for good generalization. We can see that the number of
5.2 EXPERIMENTS WITH RANKED TREE ALGORITHMS

<table>
<thead>
<tr>
<th>k</th>
<th>#documents</th>
<th>#examples (out of total)</th>
<th>Prec</th>
<th>Recall</th>
<th>F1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bigbook-name</td>
<td>5</td>
<td>10</td>
<td>40 (184)</td>
<td>100</td>
<td>98.7</td>
</tr>
<tr>
<td>Bigbook-address</td>
<td>5</td>
<td>10</td>
<td>40 (184)</td>
<td>100</td>
<td>99.1</td>
</tr>
<tr>
<td>Okra-name</td>
<td>4</td>
<td>10</td>
<td>10 (120)</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>Okra-email</td>
<td>4</td>
<td>10</td>
<td>10 (120)</td>
<td>100</td>
<td>100</td>
</tr>
</tbody>
</table>

Table 5.3: The number of examples needed for good generalization

examples needed to learn a good wrapper for these datasets is quite small.

In the ten documents of the Bigbook dataset in Table 5.3, the training time for each document ranges from 0.11 to 0.12 seconds and the average testing time for each document is 11.24 seconds (the average number of nodes in a document ≈ 513). In the ten documents of the Okra dataset in Table 5.3, the training time for each document ranges from 0.26 to 0.54 seconds and the testing time for each document varies from 0.08 to 38.67 seconds (the average number of nodes in a document ranges from 66 to 1299 respectively).

The second part of the experiment is to test the quality of the learned automata on unseen examples. The remaining data in the second part consists of 225 (= 235 - 10) and 242 (= 252 - 10) documents from the Bigbook and Okra datasets respectively. Using the learned tree automaton from the first part of the experiments, we perform extraction on the remaining documents in the data set. Note that none of these documents was used during the learning. Table 5.4 shows the results. The results for the Okra-name and Okra-email datasets are very good considering that it used only 10 examples for learning. However, the results for the Bigbook-name and Bigbook-address datasets are not as good. The reason is that the tree automaton is sensitive to the small variability in the document tree, even after generalization. In the bigbook data, there is an index in every page that enable the user to ‘jump to’ the first company name beginning with a certain letter. This index sometimes contains full links but sometimes only partial links. The variability in this bottom part of the document is creating new states that were not seen before by the tree automaton, causing failure of the extraction task.

<table>
<thead>
<tr>
<th>k</th>
<th>#documents</th>
<th>#fields</th>
<th>Prec</th>
<th>Recall</th>
<th>F1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bigbook-name</td>
<td>5</td>
<td>225</td>
<td>4115</td>
<td>100</td>
<td>70.5</td>
</tr>
<tr>
<td>Bigbook-address</td>
<td>5</td>
<td>225</td>
<td>4115</td>
<td>100</td>
<td>71.7</td>
</tr>
<tr>
<td>Okra-name</td>
<td>4</td>
<td>242</td>
<td>3214</td>
<td>100</td>
<td>97</td>
</tr>
<tr>
<td>Okra-email</td>
<td>4</td>
<td>242</td>
<td>3214</td>
<td>100</td>
<td>97</td>
</tr>
</tbody>
</table>

Table 5.4: The test on the rest of the larger datasets
Table 5.5: Comparison to the other IE systems on the larger datasets

<table>
<thead>
<tr>
<th></th>
<th>Bigbook (4 fields)</th>
<th>Okra (6 fields)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Accuracy</td>
<td>#Examples</td>
</tr>
<tr>
<td>Stalker</td>
<td>97%</td>
<td>8</td>
</tr>
<tr>
<td>WIEN</td>
<td>100%</td>
<td>274</td>
</tr>
<tr>
<td>SoftMealy</td>
<td>100%</td>
<td>6</td>
</tr>
<tr>
<td>WL²</td>
<td>100%</td>
<td>6</td>
</tr>
<tr>
<td>g1-testable</td>
<td>71.7%</td>
<td>(2 fields)</td>
</tr>
</tbody>
</table>

Hsu and Chang (Hsu and Chang 1999) list the performance of Stalker, WIEN, and SoftMealy systems on these datasets as shown in Table 5.5. In the Bigbook dataset, Stalker (Musles, Minton, and Knoblock 2001) achieves 97% recall (or accuracy in their definition) with 8 examples, WIEN (Kushmerick 1997) achieves 100% recall using an average of 15 documents containing approximately 274 examples, and SoftMealy (Hsu and Chang 1999) achieves 100% recall given 6 examples. In the Okra dataset, Stalker achieves 97% recall (or accuracy in their definition) with only 1 example, WIEN achieves 100% recall using an average of 3.5 documents containing approximately 46 examples, and SoftMealy achieves 100% recall given 1 example. Recently, (Cohen, Hurst, and Jensen 2002) compared the performance of their system, WL², to those of WIEN and Stalker. We include the results of WL² on the Bigbook and Okra datasets in Table 5.5. In fact, WL² has also been tested on the the IAF and QS datasets and is able to extract all four tasks in the IAF and QS datasets with 100% recall (or accuracy in their definition).

These results cannot be compared rigorously with ours because the above wrapper systems extract 6 fields from the Okra dataset and 4 fields from the Bigbook dataset, while our system was only tested on 2 fields from each dataset. Still, the comparison makes clear that the g1-testable algorithm needs more examples to learn from than the string-based methods we compare with. This is not unexpected: our tree-based methods search a larger hypothesis space, looking also for patterns further away from the field to be extracted, whereas the other methods look for patterns that narrowly enclose this field. As our methods consider more possible hypotheses, they need more examples to eliminate the incorrect ones.

5.2.3 Error analysis

Our ranked tree automaton inference algorithms tested in Section 5.2 exhibit only a few false negatives in Table 5.1 and Table 5.4 but no false positives, except in the small Shakespeare dataset. This means that our ranked tree algorithms extract the target fields precisely but not all of these target fields
are extracted. In other words, the learned tree automaton rejects some trees with the correctly labeled target fields. Some of the possible reasons for this are the following.

1. As mentioned previously, a tree automaton is sensitive to the small variability in the order of data locations in the document tree, even after generalization. For example, a tree with $n$ children can have $n!$ different order. The variability in the document is creating new states that were not seen before by the tree automaton, causing failure of the extraction task.

2. Related to the variability problem above and to be discussed in more detail in Section 5.4.1 is the unknown labels problem, where the unseen test set contains trees with unseen labels. The learned automaton rejects trees with labels not seen during learning.

3. Our ranked tree algorithms learn from positive examples only. Hence, in a way they are biased to generalize pessimistically. The generalization with wildcards that we introduced for the tree automaton is a way to reduce this bias. Another thing is that the criterion we use, the F1 score, results in a tree automaton that favors precision over recall.

4. Our ranked tree algorithms learn a local tree automaton, which are biased toward finding local patterns in the trees. Thus if the patterns that we want to learn are not local, the extraction ability of the learned tree automaton is rather limited because local tree automaton can not express global patterns.

5.3 Experiments with the unranked tree algorithm

The local unranked tree automaton inference method, which is used in this section, is described in Section 4.6 of Chapter 4. We evaluate this local unranked tree automaton inference method in two different ways:

- In the first experiment, we evaluate our method on some benchmark data sets commonly used in IE research. We use the same setting as used by some previous work on wrapper induction in order to provide a close comparison.

- In the second part, the experiment focuses on the efficiency of the unranked algorithm and the results are compared with those from an analysis of the algorithm based on the probably approximately correct (PAC) framework.
5.3.1 Experiment on benchmark datasets

5.3.1.1 The datasets

In the first experiment, we evaluate our method on the same datasets as used in Section 5.2.1. These datasets are:

- The Internet Address Finder (IAF) dataset: a collection of web pages containing people's contact addresses,
- The Quote Server (QS) dataset: a collection of web pages about stock quotes, and
- The small Shakespeare dataset: a significantly reduced Shakespeare XML dataset.
- The full Shakespeare XML dataset.

5.3.1.2 The results

The training and the testing processes follow the procedures outlined in Section 4.6 of Chapter 4. We use the same criteria that are used in Section 5.2.1 for evaluating our method: precision $P$, recall $R$, and the F1 score. The precision, recall and F1 scores are calculated for each extracted slot, without collecting the different slots in a case frame.

The tests on the IAF and QS datasets are performed with 10-fold cross validation following the splits used in (Freitag and Kushmerick 2000), where each split has 5 documents for training and 5 for testing. The experiment on the small Shakespeare dataset is using 2-fold cross validation, where each split has 5 documents for training and 5 for testing. The experiment on the full Shakespeare dataset is using 10-fold cross validation and each split contains 25 random examples for learning and the remaining 23 examples for testing.

Table 5.6 shows the best results of the unranked method with a certain $k_c$ that were obtained by cross-validation on one fold of 50% random training and test examples. For convenience, we include some results from Table 5.1 into Table 5.6. As can be seen, our method performs optimal and better in all these test cases than the existing state-of-the-art string-based methods and the $k$-testable tree automaton inference method. A distinguishing context was used in the datasets IAF-alt-name and IAF-organization.

Table 5.7 shows the parameters $k$ and $k_c$ that were used by the $k$-testable and our unranked tree inference algorithm respectively to produce the results in Table 5.6. It is well-known that when learning from positive examples only, there is a problem of over-generalization. The unranked algorithm, similar to the ranked tree automaton inference algorithms tested in the previous section, requires a cross-validation on the value of $k_c$ to avoid over-generalization.
5.3  Experiments with the Unranked Tree Algorithm

<table>
<thead>
<tr>
<th>IAF-all name</th>
<th>IAF-organization</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Prec</td>
</tr>
<tr>
<td>HMM</td>
<td>1.7</td>
</tr>
<tr>
<td>Stalker</td>
<td>100</td>
</tr>
<tr>
<td>BWI</td>
<td>90.9</td>
</tr>
<tr>
<td>k-testable</td>
<td>100</td>
</tr>
<tr>
<td>unranked</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td>Prec</td>
</tr>
<tr>
<td>HMM</td>
<td>36.3</td>
</tr>
<tr>
<td>Stalker</td>
<td>0</td>
</tr>
<tr>
<td>BWI</td>
<td>100</td>
</tr>
<tr>
<td>k-testable</td>
<td>100</td>
</tr>
<tr>
<td>unranked</td>
<td>100</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Small Shakespeare</th>
<th>Full Shakespeare</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prec</td>
<td>Rec</td>
</tr>
<tr>
<td>HMM</td>
<td>56.2</td>
</tr>
<tr>
<td>Stalker</td>
<td>100</td>
</tr>
</tbody>
</table>

Table 5.6: Comparison of the results

Table 5.7: Parameters used for the experiments

5.3.2  Efficiency of the unranked algorithm

The goal of the second experiment, described in this subsection, is to evaluate the efficiency of the local unranked algorithm. We first analyze the number of examples needed for effective generalization. Then we analyze the time needed for training given some datasets. Our prototype implementation was implemented in Prolog without much attention paid to optimizations and was tested on a Pentium IV 1.7 GHz PC.

5.3.2.1  The datasets

For the experiments in this section, we use the same datasets, which are also used in Section 4.5. These datasets consist of a total of eight tasks from the IAF, QS, BigBook, and Okra datasets.
<!-- DTD for Shakespeare  J. Bosak  1994.03.01, 1997.01.02 -->
<!-- Revised for case sensitivity 1997.09.10 -->
<!-- Revised for XML 1.0 conformity 1998.01.27 (thanks to Eve Maler) -->

<!ELEMENT PLAY    (TITLE, FM, PERSONAE, SCNDSCR, PLAYSUB, INDUCT?, PROLOGUE?, ACT+, EPILOGUE?)>
<!ELEMENT TITLE   (#PCDATA)>
<!ELEMENT FM       (P+)>
<!ELEMENT PERSONAE (TITLE, (PERSONA | PGROUP)+)>
<!ELEMENT PGROUP   (PERSONA+, GNPDSCR)>
<!ELEMENT PERSONA  (#PCDATA)>
<!ELEMENT GNPDSCR  (#PCDATA)>
<!ELEMENT SCNDSCR  (#PCDATA)>
<!ELEMENT PLAYSUB  (#PCDATA)>
<!ELEMENT INDUCT  (TITLE, SUBTITLE*, (SCENE+|(SPEECH|STAGEDIR|SUBHEAD)+))>
<!ELEMENT ACT     (TITLE, SUBTITLE*, PROLOGUE?, SCENE+, EPILOGUE?)>
<!ELEMENT SCENE   (TITLE, SUBTITLE*, (SPEECH | STAGEDIR | SUBHEAD)+)>
<!ELEMENT PROLOGUE (TITLE, SUBTITLE*, (STAGEDIR | SPEECH)+)>
<!ELEMENT EPILOGUE (TITLE, SUBTITLE*, (STAGEDIR | SPEECH)+)>
<!ELEMENT SPEECH  (SPEAKER+, (LINE | STAGEDIR | SUBHEAD)+)>
<!ELEMENT SPEAKER (#PCDATA)>
<!ELEMENT LINE    (#PCDATA | STAGEDIR)>
<!ELEMENT STAGEDIR (#PCDATA)>
<!ELEMENT SUBTITLE (#PCDATA)>
<!ELEMENT SUBHEAD (#PCDATA)>

Figure 5.6: A DTD of the full Shakespeare dataset

Additionally, we also test the local unranked algorithm using the full Shakespeare dataset, where previous tree-based methods, such as k-testable, g-testable and gl-testable methods, perform poorly, because a large structural context is needed to extract the fields of interest. In this dataset, the task is to extract the title of the second scene from every act in a particular play. The DTD of the full Shakespeare dataset is shown in Figure 5.6. As we can see from its DTD, this dataset has a quite complex structure, and this makes the extraction task challenging.

5.3.2.2 Sample efficiency
We use the following setting to perform the tests. For each extraction task, we gathered a set of examples that is taken from ten documents in the dataset. From each document, we marked the fields of interest to get several examples. Note that the number of examples per document differs from one document to
another. The total number of examples for all ten documents in all datasets is shown in Table 5.8 (between parenthesis in the fourth column). From this set of examples we randomly choose five examples, then ten examples, then fifteen, and so on, and give them as training examples to our learning algorithm while the rest is for testing. This process is performed 10 times for every extraction task. We stop when the learned automaton has an average F1 score of 95% or better on the test examples. Table 5.8 (fourth column) shows the number of examples the learning algorithm needed to reach that score. From this experiment, we can conclude that the number of required examples to learn an effective wrapper is quite small.

Figure 5.7 shows the evolution of F1 score, precision and recall with the number of examples on the full Shakespeare dataset with $k = 3$. As we can see, the unranked tree automaton inference algorithm is very precise and the recall can be improved with additional number of examples.

The above experiment can be considered as providing an empirical investigation of the sample complexity of our tasks. In the following, we pursue a more theoretical investigation of the sample complexity. Specifically, we analyze our method with the PAC framework (Valiant 1984), see e.g. (Angluin 1992; Kearns and Vazirani 1994) for a survey. Informally, the goal of the PAC model is to characterize classes that can be reliably learned given a reasonable number of training examples with a reasonable amount of computation.

Consider a tree automaton inference problem where the learner $L$ wants to learn a target tree automaton $M$. Given a set of training examples $\varepsilon = \{t_1, \ldots, t_n\}$, that is drawn from an unknown probability distribution $D$, the learner’s task is to generate an output hypothesis $h \in H$ that approximates $M$. Here we are interested in how closely the learner’s output hypothesis $h$
approximates the actual target automaton $M$. For this purpose, the PAC model uses an error measure, which is called true error. True error is defined as the probability that $h$ will misclassify an instance drawn at random according to $D$:

$$\text{error}(h) \equiv Pr_{p \in D}[h(p) \neq M(p)]$$  \hspace{1cm} (5.1)

Ideally, we want $h = M$ or $\text{error}(h) = 0$. In general, this is futile in the setting we are considering because the training examples that are drawn randomly might be misleading. To accommodate this difficulty, the PAC model requires that with probability at least $(1 - \delta)$ the learner will output a hypothesis $\hat{h}$, such that the true error $\text{error}(\hat{h})$ is bounded by some constant $\epsilon$ that can be made arbitrarily small.

Figure 5.7: The learning curve of the unranked algorithm
5.3. EXPERIMENTS WITH THE UNRANKED TREE ALGORITHM

Our learners are consistent ones as they build automata that accept all training examples; also, the hypothesis spaces are finite. For consistent learners with finite hypothesis spaces, there exist a well-known bound on the sample complexity (Mitchell 1997) (see also Theorem 2.3):

$$|e| \geq \frac{1}{\epsilon} (ln|H| + ln(1/\delta))$$

(5.2)

The inequality shown in Equation 5.2 gives a general bound on the number of training examples $|e|$ that ensures the hypothesis is probably approximately correct with parameters $\epsilon$ and $\delta$. In the formula, $|H|$ is the number of possible hypotheses.

Let us analyze $|H|$ for a local unranked tree automaton. Such a local unranked tree automaton has transitions that are in the form of $v(e) \rightarrow v$. The regular expression $e$, in each label $v$, is represented by $k$-grams$^4$ from the label $V$. The number of possible $k$-grams in each transition is equal to $|V|^k$ (including the extra symbol $\#$). In what follows, we denote $|V|$ with $V$ for brevity. Each different set of $k$-grams corresponds to a DFA. Thus, the number of possible DFA representing $e$ is equal to the number of possible subsets of the set of all $k$-grams, which is $2^{V^k}$. As there is a DFA for each label from $V$, the number of possible local unranked tree automata $|H|$ is equal to $2^{V^k}$. Note that we keep $V^k$ in the exponent for empirical reason (see below). Substituting this $|H|$ into Equation 5.2 yields the following bound for the sample complexity of learning a local unranked tree automaton:

$$|e| \geq \frac{1}{\epsilon} (V^k \ln 2 + \ln(1/\delta))$$

(5.3)

We can see from Equation 5.3 that $|e|$ grows polynomially in $V$ (for a given $k$), linearly in $1/\epsilon$, and logarithmically in $1/\delta$. Note that the values of $V$ and $k$ are independent of the number of training examples. Note also that this bound for the sample complexity does not take into account the distinguishing context that might be added as a hint to the learner. This is because the distinguishing context plays a different role than the training examples, and is related to representational issues (see also Section 5.4.2).

For the HMTL datasets and the Shakespeare XML dataset, the maximum size of $V$ ($V$) is equal to 182 ($= 91 \times 2$) and 42 ($= 21 \times 2$) respectively. We multiply the label by two to account for the convertLabel function, which might change the label by adding suffix '.x'. Actually in practice, the $V$ in the HMTL datasets and the Shakespeare XML dataset is less than the real $V$ mentioned above. This is because firstly, not all $V$ are used in the transitions and secondly, not all transitions have the same number of children, especially

$^4$The value of $k$ here is actually the value of $k_e$ in Algorithm 6
for the leaf labels. In order to get a better estimate of the sample complexity, we use the empirical values of $\mathcal{V}$ and $\mathcal{V}^{k}$, instead of using their real values.

In Table 5.9, $|\varepsilon|$ gives the bound obtained with Equation 5.2 for $\delta = \epsilon = 1/20$ and using the average number of $k$-grams in the input data as $\mathcal{V}^{k}$. As we can see the PAC bounds overestimate the number of examples needed that we get from empirical results in the fourth column in Table 5.8. Actually, it is well-known that the bound in Equation 5.2 can be an overestimate (Mitchell 1997).

| $k$ | $\mathcal{V}$ | $\#k$-grams as children | $|\mathcal{V}|$ in the children | $|\varepsilon|$ |
|-----|---------------|--------------------------|-------------------------------|------------|
| 2   | $a_{x}$       | 2                        | 2                             | 171        |
| 2   | $font_{x}$    | 2                        | 2                             | 171        |
| 2   | $html_{x}$    | 3                        | 3                             | 434        |
| 2   | $td_{x}$      | 2                        | 2                             | 171        |
| 3   | $body_{x}$    | 13                       | 7                             | 33345      |
| 3   | $form_{x}$    | 13                       | 8                             | 56843      |
| 3   | $table_{x}$   | 12                       | 3                             | 1183       |
| 3   | $tr_{x}$      | 6                        | 3                             | 1183       |

Table 5.10: Sample complexity of the Bigbook-name task

In fact, one could calculate the size of the alphabet $\mathcal{V}$ for each symbol $v$: the number of different symbols that occurs as child of $v$ in the examples used for learning. A string automaton is inferred only for a subset of $\mathcal{V}$ (symbols $v_{x}$). Also the size of $k$ ($k = 2$ is selected when not all sequences have length $k_{c}$) reduces the real complexity. Another thing that reduces the practical complexity is the decision to use "+" as the automaton. So, a more realistic table would be obtained by looking at the symbols for which a string automaton is
computed, and, for the used $k$ values to calculate the number of examples that is needed by that PAC formula. Because $V$ varies, it means one could have a whole table for each data set separately. So for data sets with $k_c > 2$ it should be split in two parts: the value for 2 and the value for the chosen $k_c$ value). As examples, we select the Bigbook-name and Bigbook-address tasks to calculate the bounds.

In Table 5.10 and Table 5.11, $|\varepsilon|$ gives the bounds obtained with Equation 5.2 for $\delta = \epsilon = 1/20$, for each symbol ($v_x$) where a string automaton is computed in the Bigbook-name and Bigbook-address tasks respectively. In the Bigbook-name, $k_c = 3$ was used. So we divide the Table 5.10 into two parts: the upper part for the $k_c$ value 2 and the lower part for the $k_c$ value 3. We can see that these bounds also overestimate the number of training examples needed. The bounds of the Bigbook-name and Bigbook-address tasks in Table 5.9 are better than the maximum $|\varepsilon|$ bounds in Table 5.10 and in Table 5.11 because they are calculated using the average numbers of $V^n$.

### 5.3.2.3 Inference process efficiency

We use the same experimental setting as described in Section 5.3.2.2 to measure the efficiency of the inference process of Algorithm 6. The fifth column of Table 5.8 shows the average induction time per example in seconds needed to learn an automaton for the corresponding datasets. The total time needed can be obtained by multiplying the average induction time and the number of examples in the column next to it. These results suggest that our method runs quite fast, all requires less than a second per example, except on the full Shakespeare dataset. The last column of Table 5.8 shows the average number of nodes per example that gives an approximate measure of the size of the documents in each dataset. A more formal analysis of the computational complexity of Algorithm 6 is described below.

As mentioned above, Algorithm 6 consists of two for-loops. In the first part including the first for-loop, it calls the convert_labels function, collects
states and transitions from roots, forks and subtrees, and finally collects the forks into partitions. The convert_labels function runs in time $O(n)$, where $n$ is the total size of the training examples. By using a suitable data structure for looking up forks and subtrees, collecting the states from 1-roots, 2-forks and 1-subtrees, and putting the forks into a partition can be done in time $O(n)$. In the second for-loop, Algorithm 6 calls the $k$-contextual algorithm for all forks in the partition that are labeled with $v_a$ and derives the transitions from these forks. The $k$-contextual algorithm, which dominates the run time in this for-loop, runs in time $O(m)$, where $m$ is the total length of all strings from the nodes’ children in the forks’ partition. In combination, the Algorithm 6 runs in time $O(n + m)$. If we assume $m \approx n$ then the time complexity becomes $O(cn)$, where $c$ is a small constant. Using similar analysis, the time complexity of Algorithm 9 is also $O(cn)$. Thus we can conclude that both Algorithm 6 and Algorithm 9 run in time polynomial with the size of the training examples.

Also from equation 5.3, we can see that $|e|$ grows polynomially in $V$ (given a limited range of $k$), linearly in $1/\epsilon$, and logarithmically in $1/\delta$. Thus, we can conclude that the class of local unranked tree automata is PAC learnable using Algorithm 6 and Algorithm 9.

## 5.4 Additional experiments

### 5.4.1 Test on the Syllabus dataset

In this subsection, we describe an additional experiment on a Syllabus dataset\(^5\). This dataset is created from the syllabus of Master in Artificial Intelligence (MAI) program (www.mai.kuleuven.ac.be) of the Katholieke Universiteit Leuven in Belgium. The idea of using this dataset for IE experiment emerges from the need to extract the duration of the teaching time in a particular course for a particular professor.

The extraction tasks that we experimented with in this dataset are: extracting the syllabus number, called number field, and extracting names of the professors, called professor field. We do not use distinguishing contexts for this experiment. Table 5.12 shows the results of the $k$-testable algorithm on the syllabus number field and the results of the unranked algorithm on both fields. We used the optimal $k$ and $k_e$ values, which are 2 for both algorithms. They were determined with cross-validation on one fold of the examples. All of these experiments were done with 10-fold cross validation.

The improvement of the results of the unranked algorithm over the $k$-testable algorithm can be attributed to the use of wildcards for generalizing its transitions.

---

\(^5\) Available from http://www.cs.kuleuven.ac.be/~mi/ie/
### 5.4. ADDITIONAL EXPERIMENTS

<table>
<thead>
<tr>
<th>Syllabus-number</th>
<th>#example</th>
<th>k/k_c</th>
<th>Precision</th>
<th>Recall</th>
<th>F1</th>
</tr>
</thead>
<tbody>
<tr>
<td>k-testable</td>
<td>11(21)</td>
<td>2</td>
<td>100</td>
<td>30</td>
<td>46.2</td>
</tr>
<tr>
<td>unranked</td>
<td>5(21)</td>
<td>2</td>
<td>100</td>
<td>28.1</td>
<td>43.9</td>
</tr>
<tr>
<td>unranked</td>
<td>10(21)</td>
<td>2</td>
<td>100</td>
<td>41.8</td>
<td>59</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Syllabus-prof</th>
<th>#example</th>
<th>k/k_c</th>
<th>Precision</th>
<th>Recall</th>
<th>F1</th>
</tr>
</thead>
<tbody>
<tr>
<td>k-testable</td>
<td>11(25)</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>unranked</td>
<td>5(25)</td>
<td>2</td>
<td>100</td>
<td>26.5</td>
<td>41.9</td>
</tr>
<tr>
<td>unranked</td>
<td>10(25)</td>
<td>2</td>
<td>100</td>
<td>42.7</td>
<td>59.8</td>
</tr>
</tbody>
</table>

Table 5.12: Results on the Syllabus dataset

From the results in Table 5.12, we can see that extracting this dataset turns out to be difficult for our tree-based methods. From the observation of tree structures of this dataset, we found that in one of the subtrees, a v_z node occurs with many siblings (we counted more than 30) and the order in which these siblings occur is not regular but permuted. Actually, our tree learners require quite a number of examples for this kind of problems to get a good performance. In fact, the problem can be solved by using this rule: find the first h2 tag in the document and the field that should be extracted is the first text after this h2 until a br tag. We guess that this rule can be learned by a string-based method with fewer examples.

In this dataset, we also observe that some labels (HTML tags) occur in one document but not in the other documents. This is because the professors use their own expertise of HTML to create the prescribed look and there is no common template followed by everyone. This situation complicates the problem further. Note that string-based approaches will also be affected by this problem if unknown labels occur as delimiters of the target field.

We conclude that when the identification of target fields does not require dependencies between nodes in the tree, but instead rely on local patterns (e.g., the field to be extracted is always surrounded by specific delimiters), our tree-based methods need more examples to learn a particular extraction rule than methods that directly focus on local patterns. Intuitively, more variations in further-away nodes need to be observed before these variations are considered irrelevant. This is simply an instance of the well-known trade-off between the generality of a hypothesis space and the efficiency with which the correct hypothesis can be extracted from it.
5.4.2 Experiment without distinguishing context

In order to show the importance of a distinguishing context for our methods, this subsection discusses the experiments without distinguishing context. Table 5.13 shows the results of the gl-testable algorithm on the IAF data without distinguishing context with $k$ value varied from 4 to 6.

Compared to the results of the gl-testable algorithm in Table 5.1, we can see that the results without distinguishing context in Table 5.13 are significantly worse. We can observe that with $k$ value equals to 4, if we compare Table 5.1 to Table 5.13, the recall stays equal while the precision drops significantly. As we already know and can be seen in Table 5.13, increasing the value of $k$ also increases the precision while at the same time decreases recall because the automaton becomes more specific. The need for a distinguishing context is related to representational issues. Recall that in the previous chapter we abstract text data at the leaves into CDATA.

We have also tested the unranked algorithm without a distinguishing context on the IAF dataset and obtained similar results, which are worse than the results obtained with the same algorithm in Table 5.6. Thus, the distinguishing context is important for our methods especially to learn from datasets such as IAF, where the abstraction of text content into CDATA eliminates the clue to indentify target fields correctly.

A distinguishing context plays a different role than the training examples. Indeed, our methods are better with $n$ examples and one distinguishing context than with $n + 1$ examples and without any distinguishing context. We can say that the additional distinguishing context acts as additional hint for our methods.

5.4.3 Experiment with noisy examples

In this subsection we describe experiments to evaluate the robustness of our methods when given noisy training examples. There are several types of noise in the examples: labeling errors, changes in the content and the document structure, etc.

Regarding the changes in the content, we can say that our methods are more
Table 5.14: Results in the small Shakespeare dataset with noisy examples

<table>
<thead>
<tr>
<th>error rate</th>
<th>k-testable, $k = 3$</th>
<th>gl-testable, $k = 4$</th>
<th>unranked, $k = 3$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Prec</td>
<td>Rec</td>
<td>F1</td>
</tr>
<tr>
<td>0% labeling error</td>
<td>56.2</td>
<td>90</td>
<td>68.2</td>
</tr>
<tr>
<td>20% labeling error</td>
<td>19.1</td>
<td>65</td>
<td>29.5</td>
</tr>
<tr>
<td>40% labeling error</td>
<td>13.9</td>
<td>100</td>
<td>24.4</td>
</tr>
</tbody>
</table>

robust than string-based methods that rely on the use of delimiters surrounding the target field. This is because our methods do not really care about the contents, which are regarded as CDATA. Unknown internal labels such as those that occur in the Syllabus dataset, however, deteriorate the performance of our methods. Our algorithms infer tree automata that learn from and accept a set of trees that are marked with \( z \), where the locations of \( z \) can be anywhere at tree leaves. In this respect, our algorithms are robust to some structural variations of \( z \).

On the other hand, if the tree structure changes completely then the tree automaton simply will reject the tree. We expect string-based methods to be more robust to cope with this kind of noise. Note that because our methods learn a tree automaton that accepts or rejects the whole document tree, in general, our methods are less robust to noise than methods that work locally such as string-based methods.

Below, we describe an experiment where labeling errors occur in the training phase. Since our methods rely on humans, instead of hand made recognizers or an oracle, to label a relatively small number of target fields, we expect that labeling errors will occur rarely. Table 5.14 shows the results of some algorithms on the small Shakespeare dataset with 20% and 40% error rate, which are quite large, in labeling. For ease of comparison, we also include the results with 0% labeling error from Table 5.1 and Table 5.6. Figure 5.8 shows graphically the F1 scores of some algorithms on the small Shakespeare dataset with 0%, 20% and 40% error rate.

Instead of the correct label, which is the title of the second scene, the labeling errors are combinations of the following (see the DTD in Figure 5.1):

- labeling \( z \) as the title of another scene than the second one,
- labeling \( z \) as the SPEAKER element, and
- labeling \( z \) as the LINE element.

We can see from Table 5.14 and Figure 5.8 that the F1 score deteriorates rapidly with labeling errors. However, in practice these errors are quite rare and the user can always correct the labeling if the error occurs. We can also
see that the unranked and gl-testable algorithms are more robust to noise than the k-testable algorithm; and the unranked algorithm is more robust to noise than the gl-testable algorithm. Notice also that the recall performance of the unranked algorithm does not deteriorate with labeling errors. Only the precision of the unranked algorithm is significantly affected.

![Small Shakespeare dataset](image)

**Figure 5.8:** The performance of the algorithms with noisy examples

Recall from Section 3.3.3 that when a noisy example is learned, then a learning method that learns from positive examples only cannot undo it. Since our methods learn from positive examples only, they are generally very sensitive to noisy examples. One way to improve the methods to cope with noise is to use probabilistic inference methods, which are known to be more robust with respect to noisy examples.
5.5 Discussion on ranked vs unranked tree automata

In Section 2.7.1, we have described the relation between ranked and unranked tree automata. Neven (1999) shows the correspondence between ranked and unranked tree automata. However, in the case of local tree languages, local unranked tree languages seem to be more expressive than local ranked tree languages. To show this, consider the following example.

![Diagram of trees](image)

**Figure 5.9:** Two trees and their corresponding binary trees

Figure 5.9 shows two trees and their corresponding binary trees. The trees can be specified as follows. A tree is labeled with $a$ if its children are labeled with any number of $b$ followed by any number of $c$. A tree is labeled with $d$ if its children are labeled with any number of $b$ followed by any number of $e$.

It is easy to see that for any $k < l$, where $l$ is the number of $b$-node, $k$-testable tree languages are not able to specify the trees in the above example correctly. As we know, using a large $k$ will result in a very specific tree automaton. Thus, it is likely that the value of $k$ is less than the value of $l$ for examples with long sequences of $b$-nodes. Local unranked tree languages, however, are able to
specify the example trees with any number of \textit{b}-nodes using $k_c = 2$.

This example provides an explanation of why local unranked tree induction
often works better than local ranked tree induction.

\section{The advantages and limitations of our methods}

Our (semi-automatic) tree-based methods have several advantages over semi-
automatic string-based methods. These are as follows:

\begin{itemize}
\item String-based methods learn string languages while our methods learn
  more expressive tree languages.
  \begin{itemize}
  \item The extracted field can depend on its structural context in a doc-
    ument. Some structural context that is close to the target field in
    the tree structure of the document can be arbitrarily far away in
    the string representing the document. This makes the learning task
    very difficult for the string-based methods and results in wrappers
    with rather poor performance.
  \item Compared to WI (Kushmerick 2000a) or SoftMealy extractors in
    (Hsu and Chang 1999), our methods are generally applicable to any
    type of document formatting without requiring different classes of
    wrappers for different document formats.
  \end{itemize}
\item Our methods require minimal user intervention and are easy to apply.
  The user only needs to label a relatively small number of examples to
  train the extractor. While some other manual pre-processing steps are
  often needed for other methods to learn the extractor.
  \begin{itemize}
  \item Compared to Whisk (Soderland 1999), our methods do not require
    splitting the document into small fragments and selecting some of
    them to be used as training examples.
  \item Compared to HMMs (Freitag and McCallum 1999) and BWI (Freitag
    and Kushmerick 2000), our methods do not require manual specifica-
    tion of windows length for the prefix, the suffix and the target
    fragments.
  \item Compared to Stalker (Muslea, Minton, and Knoblock 2001) and
    BWI (Freitag and Kushmerick 2000), our methods do not require
    manual specification of special tokens or landmarks such as ‘>’ or
    ‘\textendash’.
  \end{itemize}
\end{itemize}
5.6. THE ADVANTAGES AND LIMITATIONS OF OUR METHODS

- Compared to Stalker (Muslea, Minton, and Knoblock 2001), our methods do not require 13 ordering heuristics to learn disjunctive extraction rules. Instead, our methods require ‘hints’ in form of distinguishing context.

- Our methods are efficient in the sense that the time needed to learn the extractor is short. Actually, our algorithms, with the exception of the gl-testable algorithm, can learn more quickly than some string-based methods.

Despite the advantages mentioned above, our methods also have some limitations compared to semi-automatic string-based methods. These are as follows,

- Our methods only output a whole node, which might appear to limit their applicability. Certainly this could be true for HTML-formatted documents. However, for data-centric documents such as XML documents, the applicability is not really limited because the data to be extracted is typically a whole node. One way to broaden the applicability of our methods is to perform a two step extraction. In the first step, a whole node in a tree can be extracted, and in the second step other techniques can be employed to post-process the selected node in order to extract a part of it. Another possible solution is to view each word or any subsequence of texts at the leaves as a unique node that is a part of document tree.

- Our methods usually require more examples to learn a comparable wrapper. This situation typically occurs when the identification of target fields does not require dependencies between nodes in the tree but can rely on a local pattern (e.g., the field to be extracted is always surrounded by specific delimiters). In this case, our tree based methods need more examples to learn a particular extraction rule than methods that automatically focus on local patterns.

- Our methods are possibly slower during extraction than string-based methods because they have to parse the document tree and have to substitute each node with \( x \) when extracting information from the document.

- Our methods only output a single field, or slot, in one run.

- Our methods only work on structured documents. Indeed our methods cannot be used for text-based IE, and are not intended for it.

Some possible ideas for extensions to deal with the limitations of our methods will be discussed in more detail in Chapter 7.

Despite these limitations, our results suggest that our tree-based methods work as well as or better than more generally applicable state-of-the-art string-based IE methods, in difficult structured domains. Our results also suggest
that utilizing the tree structure of the documents is worthwhile for difficult structured IE tasks.

5.7 Summary

In this chapter, we have experimented with our tree automata based methods on several datasets.

In the first part, we have experimented with three ranked tree automaton inference algorithms. The results on difficult benchmark datasets show that in general these algorithms are better than state-of-the-art string based learners. We have also shown that the two generalized algorithms are better than the original $k$-testable algorithm for some IE problems. We have also provided the details of the experiments to evaluate the characteristics of the $k$-testable algorithm and its generalization: $gl$-testable algorithm. Specifically, we have described the trade-off between precision and recall across different $k$ values. Our methods require a cross-validation on the value of $k$ as a way to avoid over-generalization. The experiments on larger but simpler datasets, such as Okra and BigBook datasets, suggest that our tree-based method requires more training examples to get a comparable performance to the string based methods. Towards the end of this part, we have described an error analysis. We have investigated why our algorithms exhibit typically only a few false negatives and which factors are at the origin of the errors.

In the second part, we have experimented with a local unranked tree automaton inference algorithm. The experiments on difficult benchmark datasets show that the performance of this algorithm is better than the state-of-the-art string based learners, and also better than our ranked tree automaton inference algorithms. In fact, this algorithm is the only one giving optimal results. The second experiments in this part were designed to study the efficiency of the algorithm and to analyze the algorithm using the PAC framework. Specifically, we have analyzed how many examples are needed by the unranked algorithm to get a good generalization and how long it takes to learn from a specific dataset. From the analysis, we have concluded that the class of local unranked tree automata is PAC learnable. The experiments and the analysis of the algorithm show that the local unranked algorithm is efficient. A notable experiment in this part is the test on the full Shakespeare dataset, which was unfeasible using the ranked tree automaton inference algorithms described in the previous chapter.

We have also performed additional experiments: experiment on the Syllabus dataset, experiment without distinguishing context, and experiment with noisy examples. The experiment on the Syllabus dataset shows that our methods need more examples to learn a particular extraction rule than methods that directly focus on local patterns. The experiment without distinguishing
context shows that the precision of our methods drops significantly on the IAF dataset. We have also pointed out that the distinguishing context is related to representational issues. The experiment with noisy examples shows that our methods are robust to changes in the context and to some structural variations of the special label $z$. However, our methods are sensitive to labeling errors.

Towards the end of this chapter, we have discussed why the unranked tree algorithm often works better than the ranked tree algorithms that we used in the experiments. We have also summarized the advantages and disadvantages of our methods.

The results in this chapter show that using the unranked tree automaton inference algorithm for structural IE yields a better performance than using the ranked ones. This is true especially when the distance between the relevant context and the target field is indeed a main factor determining the ability to learn an appropriate automaton.

On the other data sets, where the tree structure of the document is unimportant, our results are less good than those of some string-based methods. This confirms that it remains important to select a method with an appropriate bias, when inducing wrappers.

The results indicate that utilizing the tree structure of the documents is worthwhile for difficult structured IE tasks. In Chapter 6, we will characterize some structural extraction tasks based on their difficulties.
Chapter 6

Related work

6.1 Introduction

In this chapter, our goals are two-fold. Our first goal is to give a more detailed survey of IE work that is related to ours. Here we will concentrate on IE from structured data, because the focus of our research is on structured data and we want to situate our work in this rapidly growing field.

In the first section, we identify the driving forces behind IE research in general. Then, in the next section, we describe some work in IE from structured data, which is our primary focus. In this part, we characterize structured IE work from two different angles:

- the degree of user involvement, and
- whether the system considers the tree structure of the document.

We use this characterization in grouping some related work on IE from structured documents. After that, we describe briefly some related problems: wrapper maintenance, schema inference, query languages, and the larger field of information integration.

Our second goal is to analyze the current state of structured IE research. Specifically, we begin the second part by describing two observations in this research domain. Given some characteristics of the structured IE systems that have been described, we might be tempted to ask which IE method is the most suitable for a given task. This question is analyzed in the following section.

Finally we conclude the chapter.
6.2 Information extraction: an emerging field of research

Although IE research is relatively new compared to related IR research, it is now becoming a very big field on its own and is attracting many researchers from different fields of study, including researchers who work with computational linguistics, databases, computational biology, and document research. Many of the methods for solving IE problems are described in (Muslea 1999; Soderland 1999; Kushmerick 2000).

Currently there are three major trends that push the development of IE research. They are as follows:

- The demand for security and intelligence applications.
- The demand for better searching tools on the Web.
- The growing interest in bioinformatics research.

There are undoubtedly more trends pushing the development of IE research, but we will focus on these three.

One of the original key drivers of (classical) IE research is the US Department of Defense. Through the Advanced Research Projects Agency (ARPA), it set the objectives for information extraction systems and funded the development of large corpora (or datasets) in some specific domains, in order to compare IE systems in several Message Understanding Conferences (MUCs) series (Cowie and Lehnert 1996; Cardie 1997). Some early applications for information extraction systems worked on datasets of news reports with domains including mergers and acquisitions, such as shown in Figure 2.12, and on terrorism, which is a classic example used by many IE researchers. Much of the research done in conjunction with MUCs is related to unstructured free texts. Today, there is an ever increasing need for security and intelligence applications, and these push the development and practical implementation of IE systems even further.

Another important factor pushing IE research is the popularity of the Internet, and specifically of the Web as a single huge repository of data. It is generally accepted that we are currently overloaded with information (Maes 1994). It is argued, e.g. in (Kosala and Blockeel 2000), that IE would be very useful specially for those people who make intensive use of information. There are many interesting applications for IE, some of which are mentioned in Chapter 3. These include:

- Using IE to support intelligent information agents that seek information on the users’ behalf. For example, an agent is assigned to find the lowest price of a certain book in the Internet. The price information is typically
just some fragments of documents they find on the Internet. IE techniques are needed for such a purpose.

- Using IE to support companies that might be interested in gathering information from online information sources, such as online newspapers, stock quotes pages, online job portals, etc. The Internet is a rich source of information that can support companies in their decision making processes. Again, better IE techniques would help them access that information.

- Using IE to support an individual who might wish to extract specific information from specific information sources in the Internet. With improved IE techniques, users would receive only the right information, not a long list of documents which may or may not be relevant.

The examples of IE applications mentioned above are indeed very promising. Some of them are not yet fully realized. However, with the increasing interest in research in this field, IE technology might soon be able to fulfill this promise.

Biological texts present many new opportunities and challenges for the field of information extraction (Hirschman, Park, Tsuji, Wong, and Wu 2002). Most IE problems in biological texts are focused on unstructured texts. Thus many techniques developed for classical IE can be and have been applied in this domain. For several years, the Pacific Symposium on Biocomputing series (http://psb.stanford.edu/) has dedicated a special track to natural language processing and information extraction in biology. (Hirschman, Park, Tsuji, Wong, and Wu 2002) mention that the response to the call for papers and the quality of submitted papers both show that this is an emerging field. A quotation from (http://www.bionlp.org) says:

The literature of the field of biology is the largest of all the sciences. The volume of biology literature each year, measured in bytes, is about fifty times the size of the entire human genome, junk and all. But locked in this literature is an enormous amount of information that can tell us much about the structure and function of genes, proteins, cells and organisms – how they work as well as how they can fail.

The newly emergent interest in natural language processing for biology has been christened "Information Extraction".

Besides the above mentioned driving forces, information extraction is also very interesting from a computational linguistic perspective. Computational linguistics is a well-established research field and has been studied for quite a long time. Although IE (from free texts) is not equal to deep language understanding, it involves many difficult and interesting sub-problems in NLP, such
as co-reference resolution and the identification of various relations among entities in real-world texts. Indeed, classical IE could be called a core language technology (Wilks 1997). Perhaps the most important aspect of information extraction is that its task is well defined, and therefore the performance of an IE system can be concretely and automatically compared to human performance (Hunter 2000). Thus IE presents a very special opportunity for NLP researchers.

6.3 IE from structured data

In this chapter, we focus our discussion of related work on the domain of IE from structured data. The reasons for this are, firstly, because the focus of our research is on structured data and, secondly and more importantly, to better situate our research in the large and interdisciplinary IE field. Even in the structured IE domain, the amount of research is quite large and growing rapidly; several different research fields have made their contributions. Thus, there are undoubtedly some unintentional omissions in our coverage of structured IE work.

As previously mentioned in Chapter 3, there are several trends that necessitate the need for structural IE systems which extract information from (semi-) structured documents. These trends include:

- the increasing popularity of the Web as a medium for disseminating information,
- the recent work on intelligent information agents (Doorenbos, Etzioni, and Weld 1997; Green, Hurst, Nangle, Cunningham, Somers, and Evans 1997),
- the recent work on information integration from the database community (Florescu, Levy, and Mendelzon 1998; Levy, Knoblock, Minton, and Cohen 1998; Wiederhold 1996).

According to (Sahuguet and Azavant 1999b), over 80% of the information published on the Web is generated from databases, and this proportion keeps increasing. This means that most published information on the Web has a regular structure. Thus, IE systems that are able to extract from structured data could play an important role.

As mentioned previously in the IE taxonomy of Figure 3.1, the work on IE can be classified into three main categories: IE from unstructured texts, IE from semi-structured texts and IE from structured texts. Within each of these categories, the work can be further divided into manually built systems and (semi-) automatic systems. As we noted previously, there are many ways to
6.3. IE FROM STRUCTURED DATA

<table>
<thead>
<tr>
<th>Type</th>
<th>String-based</th>
<th>Tree-based</th>
</tr>
</thead>
<tbody>
<tr>
<td>Semi-automatic</td>
<td>(Hsu and Dung 1998), (Freitag and McCallum 1999), (Kushmerick, Weld, and Doensbos 1997), (Kushmerick 2000a), (Musca, Minton, and Knooblock 2001), (Freitag and Kushmerick 2000), (Soderland 1999), (Chidlovskii, Ragettli, and de Rijke 2000)</td>
<td>(Freitag 1998a), (Cohen, Hurst, and Jensen 2002), (Sakamoto, Arimura, and Arikawa 2002), (Kossl, Van den Bussche, Bruynooghe, and Blockeel 2002), (Kossl, Bruynooghe, Van den Bussche, and Blockeel 2003)</td>
</tr>
<tr>
<td>Automatic</td>
<td>(Hong and Clark 2001), (Chang and Lui 2001), (Crescenzi, Mecca, and Merialdo 2001)</td>
<td>(Hemmati and Bessen 2002), (Cohen and Fan 1999), (Cohen 1999)</td>
</tr>
</tbody>
</table>

Table 6.1: Some examples of IE systems for structured data

characterize IE work. Currently the most complete taxonomy of IE is proposed by Crespo, et al. (Crespo, Jannink, Neuhold, Rys, and Studer 2002). For the purpose of this thesis, we use the IE taxonomy in Figure 3.1. Within the domain of extraction from structured documents, we can organize the research as shown in Table 6.1.

In Table 6.1, we view the IE systems that work on the structured documents from two different angles:

1. The first angle is the degree of automation of the development process of the wrapper (or the degree of user involvement). In this view, we divide IE systems into three types: manual, semi-automatic, and automatic.

2. The second angle stems from the fact that structured documents have a tree structure. Thus it is useful to view whether the wrapper uses the tree structure of the structured documents to guide the extraction of the fields of interest. We divide the IE systems into those that employ a method that works on trees (tree-based) and those that simply view the structured document as a sequence of abstract tokens, and therefore employ a method that works on strings (string-based).
We classify an IE system as a manually-built system if the user manually crafts the extraction rules, in a low level pattern language or in a high level query language, for every set of “similar” pages that typically originate from the same web site and have the same page format. When there is a new extraction task, the user has to craft new extraction rules or queries. In basic query language systems, the user has to write queries, and this is comparable to writing extraction rules manually. In the more user-friendly query languages, the systems provide a fancy and sophisticated graphical user interface (GUI) that helps the user to write queries or wrappers. With such a fancy user interface, the user does not have to know the syntax of the query language or the syntax of HTML/XML documents in order to build the wrappers. However, the user has to invest extra effort in learning the system and in providing sufficient information to the GUI beyond simpy labeling the fields to be extracted. A characteristic of manually-built IE systems is that the development process requires non-trivial knowledge from the user.

We classify an IE system as a semi-automatically built system if the development of the system only requires the user to label the fields to be extracted as examples to the system. IE systems that are built semi-automatically typically use machine learning or related techniques that can learn to extract information from a few annotated examples. Typically a new labeling and training process is needed for each new extraction task. Compared to the manual approaches described above, this approach requires less user effort and knowledge.

We classify an IE system as an automatically built system if the wrapper is only built once, either with or without training and can be used for new extraction tasks directly, or if the wrappers are built for each new task but use unsupervised training. These two approaches are described in more detail below.

1. Some automatically built IE systems are built only once. These IE systems are typically built based on some assumptions or heuristics about the structured documents. Then the IE systems are expected to perform well on unseen documents whose structures are in accordance with the heuristics assumptions. We can further distinguish between approaches that require and do not require training.

   - In the non-training approach, typically the user builds a kind of ontology or template manually. This ontology or template can be used as background knowledge about the structure of the fields that are going to be extracted.

   - In the training approach, the user provides labeled examples as input to an algorithm that learns heuristics, which, in turn, can be used to extract the fields of interest from pages with a certain formatting.
6.3. IE FROM STRUCTURED DATA

2. Other automatically built IE systems use unsupervised learning techniques. Such systems require neither user intervention nor labeling. However, the systems need training for each new extraction task.

Of course, wrappers made using this approach assume certain general page formatting, such as pages with tables, pages with lists, etc. Wrappers made for table formatting will not work for list formatting, and vice versa. Furthermore, systems that are built automatically are usually only able to extract document blocks. This block could be a paragraph or a record. Some other heuristics have to be applied to further separate all the fields from the document blocks or records. These systems are able to extract all fields in a record or document block but are not able to extract a specific field of interest from a document.

Below, we will review some recent related IE works that are applied to structured documents and will situate them in our classification. We wish to note that our classification might be somewhat imprecise because the categories in our taxonomy are not always independent or mutually exclusive. Some systems might use mixed approaches.

We would also like to note that, although the structured IE field is relatively new and much research is still being done, there are already many commercial wrapper generator toolkits available, see e.g. in (Kuhlins and Tredwell 2002). This shows the potential demand for work in this area.

6.3.1 Manually-built structured systems

The construction of a manually-built structured IE system requires non-trivial user intervention and relatively intensive knowledge of query language and/or the system itself.

6.3.1.1 String-based

Earlier wrapper systems were developed with the purpose of integrating information from different sources on the Web and were built manually. Some examples of manually built systems can be found in (Atzeni and Mecca 1997; Hammer, Garcia-Molina, Cho, Crespo, and Aranha 1997; Huck, Fankhauser, Aberer, and Neuhold 1998). They apply knowledge engineering techniques for the building of wrappers based on pattern languages.

In Editor (Atzeni and Mecca 1997), the user can search for a specific fragment or region of interest in semi-structured data, such as the data in HTML documents, and perform cut and paste operations to move and restructure the parts of a document. To locate the region of interest, the user has to define a pattern using a regular expression.

Tsimnisis wrapper (Hammer, Garcia-Molina, Cho, Crespo, and Aranha 1997) is a configurable extraction routine that can extract a specific fragment from
an HTML document. The extraction routine is specified by a sequence of commands. Each command is of the following form: [variables, source, pattern], where ‘source’ specifies the input text, ‘pattern’ is based on a regular expression pattern to locate the text of interest within the source, and ‘variables’ are one or more variables to hold the extracted results.

Jedi (Huck, Funkhauser, Aberer, and Neuhold 1998) is a mediator and wrapper development tool designed to extract, combine and reconcile information from several information sources. Its extraction language is based on context free grammars, which are more powerful than string-based methods, and more robust with respect to ambiguous extraction.

Some other string-based systems can be found in (Florescu, Levy, and Mendelzon 1998; May and Lausen 2000). The systems mentioned above do not provide visual support for the user. Finally, note that manually building a wrapper for each data source requires expertise and becomes infeasible when confronted with the variety of web sources.

6.3.1.2 Tree-based

A separate area of study in the field of manually-built systems focuses on the development of query languages for HTML/XML documents. These languages explicitly use the tree structure of the document. Most of the work in this area originated within the database community, while some other work originated within the document and logic programming communities. Some of these query languages do not provide visual support for the user, e.g. Florid (May and Lausen 2000), and Xcerpt (Bry and Schaffert 2002). Some recent wrapper generator systems that provide GUI are W4F (Sahuguet and Azavant 1999a), XWrap (Liu, Pu, and Han 2000), and Lixto (Baumgartner, Flesca, and Gottlob 2001). Although these latter systems were proposed as wrapper generator systems, they are basically query languages in disguise.

Florid (May and Lausen 2000) is an integrated architecture for information integration systems. It provides an integrated language for mediating, wrapping, querying, and surfing the information on the Web. The integrated language is based on a deductive object-oriented language called F-logic. It provides reusable generic rule patterns for specific applications, which are refined by additional rules provided manually by the user.

Xcerpt (Bry and Schaffert 2002) is a query and transformation language for XML and semi-structured data. It is based on logic programming, which many argue is more declarative than the usual path-oriented node selection method used by most query languages. It uses a novel form of unification, which is called simulation unification, and differs from the standard unification used by most of the Prolog systems.

W4F (Sahuguet and Azavant 1999a) is based on a query language called HEL, which works on the tree structure of a semi-structured document. In
W4F, a partial query can be specified using a visual tool. However, the full query must be programmed manually.

In XWrap (Liu, Pu, and Han 2000), the system parses the HTML tree and then guesses the sections, headings and fields of interest in the HTML tree using some heuristics. The user can correct the guess if it turns out to be incorrect. The Xwrap provides a GUI for the user, which then allows the user to define some patterns in order to correct the guess and to correct the semantic mapping between the field and its semantic token. Finally, the system can generate an XML file from the HTML input file using templates.

Lixto (Baumgartner, Flesca, and Gottlob 2001) is a wrapper generator program for translating HTML into XML using a visual and interactive user interface. At the core of Lixto is the Elog declarative language. Elog is a very expressive query language for tree-based semi-structured documents: it allows for extraction based on the surrounding context, on the HTML attributes, on the order of appearance, and on semantic and syntactic concepts. Elog", a fragment of the Elog query language, has been shown to be equivalent to full monadic second order logic (MSO), a logic that is equivalent to tree automata (Gottlob and Koch 2002). Lixto also provides an automatic string-pattem generator for extraction from flat strings. In addition, Lixto supports disjunctive pattern definition, recursive wrapping, and crawling and extracting other web pages. The advantage of Lixto over W4F and XWrap is that it provides the user with a sophisticated user interface that allows him to define complex queries, and it also simplifies the wrapper specification process, so that the user does not need to have expertise in writing Elog queries or know HTML syntax.

Many more query languages have been proposed for XML (Fernandez, Siméon, Wadler, Cluet, Deutsch, Florescu, Levy, Maier, McHugh, Robie, Suciu, and Widom 1999) and semi-structured data (Florescu, Levy, and Mendelzon 1998) by the database community, but they will not be mentioned here. While these query languages are suitable for expressing complex extraction problems, their use remains time consuming and requires non-trivial skill.

There also exist some other manually-built structured IE systems that are quite different from the traditional query language approaches. One of these worth noting is a template-based structured IE system introduced by (Hsu and Yih 1997). A document template here is a tree structure that specifies the logical structure of a document, which is then typically shared among documents of the same class. For extracting information in a document, a template that covers the fields of interest has to be created. The information extraction problem is then cast as the matching between the template tree and the HTML document tree. Thus the IE problem is transformed into a tree matching problem, and many methods developed for tree matching can be used (see (Shasha, Wang, and Giugno 2002)). This system, however, requires manual specification of the template.
6.3.2 Semi automatically-built structured systems

IE systems from structured documents that are semi-automatically built typically make use of machine learning and data mining techniques, as well as other algorithms. These methods often rely on a set of labeled training examples for learning the extraction rules. Our work is situated in this domain. In the literature, the term wrapper induction is used to denote the process of learning an IE system that does not use linguistic pre-processing for structured documents. With the recent popularity of the Web, this line of work has received considerable attention. It can be noted that some more recent wrapper induction systems (not ours) are also able to work on semi-structured data and even on unstructured texts.

6.3.2.1 String-based

The term wrapper induction was first introduced in (Kushmerick, Weld, and Doorenbos 1997). As mentioned in the introduction, much work on wrapper induction concerns wrappers based on regular expressions. Some structural IE systems (wrappers) that have been applied to work with structured data are (Hsu and Dung 1998; Freitag and McCallum 1999; Kushmerick, Weld, and Doorenbos 1997; Kushmerick 2000a; Muslea, Minton, and Knoblock 2001; Freitag and Kushmerick 2000; Soderland 1999; Chidlovskii, Ragetti, and de Rijke 2000). These systems use techniques that are basically equivalent to machine learning or grammatical inference techniques to induce a kind of delimiter-based string pattern.

WIEN (Kushmerick, Weld, and Doorenbos 1997) is a wrapper induction environment that implements several classes of wrappers. In (Kushmerick 2000a), six classes of wrappers are proposed. The most basic extractor in WIEN uses only the left and right delimiters of the target fields. It also assumes that there is a unique multi-slot rule that can be used for all documents. Thus it does not allow disjunctions in a rule. It is shown that simple classes of wrappers that learn just the left and right delimiters of the target field can be more effective than classes of wrappers which are more expressive. Furthermore, a slightly more expressive wrapper class can add substantial computational burdens. These results demonstrate that, for problems that do not require expressive methods, a simpler method is preferable. This is consistent with our conclusion in the previous chapter.

BWI (Freitag and Kushmerick 2000) is essentially a boosting approach in which the weak learner learns a simple regular expression with high precision but low recall. Boosting is a technique for improving the performance of a simple learning algorithm through repeatedly re-learning the training set. The relearning is performed by giving more weight to the training examples that are difficult to learn. The weak learner learns the fore (or left) delimiters and the
6.3. **IE FROM STRUCTURED DATA**

... aft (or right) delimiters of the target field several times. Then BWI repeatedly searches for the best combination of left and right delimiters of the length that is specified by a lookahead value (or below). The procedure ends when no combination yields a better score than the current one. BWI uses several types of wildcards in its extraction rules. BWI has been tested on several datasets ranging from unstructured to structured texts and has been shown to be one of the best IE methods at this time.

The HMM approach used for comparison in our experiments in Chapter 5 was proposed by Freitag and McCallum (Freitag and McCallum 1999). They learn a Hidden Markov Model, solving the problem of probability estimation from sparse data by using a statistical technique called shrinkage. Hidden Markov Models are equivalent to finite state automata that have probabilities attached to the transitions and non-deterministic state labels in the transitions. This model has been shown to achieve state-of-the-art performance on a range of IE tasks, ranging from unstructured texts to structured data.

The Stalker algorithm (Muslea, Minton, and Knoblock 1999; Muslea, Minton, and Knoblock 2001) induces extraction rules that are expressed as simple landmark grammars, a class of finite automata. Stalker performs hierarchical extraction guided by the *embedded catalog* (EC) description of a page to be extracted. This EC description is a tree-like structure which describes the structure of the fields to be extracted from the documents. The user provides labeled examples by marking the data to be extracted using a GUI. Then Stalker learns disjunctive rules by generating an initial set of candidates and repeatedly selects and refines the best candidate until it finds a perfect disjunct or until the candidate set is empty. The output of Stalker is a grammar, called Simple Landmark Grammar, in which each path corresponds to a learned disjunct. Guided by the EC tree, the Stalker string-based extraction rules are able to perform the hierarchical extraction of arbitrary levels of embedded data. Stalker has been shown to perform well on the IE problems introduced in (Kushmerick 1997) and better than WIEN on two more difficult datasets: Internet Address Finder and Quote Server. Although this method uses a kind of tree structure, this tree structure is not the parse tree of the HTML document. Furthermore, the framework views a document as a sequence of abstract tokens (words, numbers, HTML tags, etc.). Therefore, we classify Stalker as a string-based semi-automatic approach.

WHISK (Soderland 1999) is a system that learns extraction rules with a top-down rule induction technique. The extraction rules of WHISK are based on a kind of regular expression pattern. To make the rules more powerful, WHISK has several built-in semantic classes, and in addition allows for user-defined semantic classes. A semantic class is basically a set of terms that are considered to be equivalent. WHISK has been tested on several datasets that include those with both unstructured and structured texts. WHISK requires that some...
documents be broken into multiple instances before learning, which explains its ability to learn multi-slot extraction rules. Optionally, WHISK is able to use linguistic knowledge in the form of syntactic or structural information.

Chidlovskii et al. (Chidlovskii, Ragetti, and de Rijke 2000) describe an incremental grammar induction approach: their language is based on a subclass of deterministic finite automata that do not contain cyclic patterns. The algorithm is incremental and is based on the adaptation of a string-edit distance algorithm. The system is shown to be successful when it is used as a meta search tool to extract results from some search engines.

Hsu and Dung’s SoftMealy system (Hsu and Dung 1998) learns separators which identify the boundaries of the fields of interest. These separators are expressed as finite state transducers and are described by strings of fixed height, in which each symbol is an element of a taxonomy of tokens (with fixed strings on the lowest level and concepts such as punctuation or word at higher levels). Besides allowing semantic classes, it also allows disjunctions, which are useful for documents with various orderings of target fields. Its extraction rules are more expressive than the rules learned by WIEN.

Hsu and Chang (Hsu and Chang 1999) propose two classes of SoftMealy extractors: single-pass extractors, which are biased for tabular documents such as the QS dataset, and multi-pass extractors, which are biased for tagged-list documents such as the IAF dataset. Although their systems were tested on the same datasets as our system, their results cannot be compared directly because the experimental setting is different. Their evaluation gives only the recall and uses a different set of examples.

### 6.3.2.2 Tree-based

As we know, there is an explicit tree structure in structured documents such as HTML and XML documents. However, string-based methods consider a structured document to be a string, not a tree. There have been several previous works that propose using the tree structure of the structured documents, in a manner similar in spirit to our tree automata based methods. These previous works will be described in more detail below.

SRV (Freitag 1998a) is a relational learning algorithm for information extraction. It learns from a set of extensible token oriented features. Thus the user can design many kinds of tokens, such as character type features, and part of speech features, depending on what is needed. In order to learn the structure of an HTML document, the user simply has to add several HTML-specific features, such as `table_next_col`, `table_row_header`, etc., to SRV’s basic feature set. Its extraction rules are in first-order logic and are able to express the relational structure of the documents. Freitag also shows that the incorporation of HTML features yields considerable improvement, especially for precision.
6.3 IE FROM STRUCTURED DATA

Sakamoto et al. (Sakamoto, Arimura, and Arikawa 2002) present a certain class of wrappers that use the tree structure of HTML documents, and propose an algorithm for inducing such wrappers. They identify a field with a path from root to leaf, imposing conditions on each node in the path that relate to its label and its relative position among siblings with the same label (e.g., “2nd child with label <B>”). Actually, their wrappers work in a way similar to the way XPath (discussed in a later section) chooses some nodes in the document tree. Their hypothesis language corresponds to a subset of tree automata.

WL2 (Cohen, Hurst, and Jensen 2002) is a logic-based wrapper learner that uses multiple (string, tree, visual, and geometric) representations of HTML documents. It uses a modular learning architecture, which consists of one master learner and several “builders” which are adaptable to specific extraction tasks. The master learner is based on Foil (Quinlan 1990), an inductive logic programming (ILP) algorithm. In fact, WL2 is able to extract all four tasks in the IAF and QS datasets with 100% recall. The work of WL2 suggests that indeed the use of task-specific document representation can yield better performance than the string-based learners used previously.

6.3.3 Automatically-built structured systems

6.3.3.1 String-based

Hong and Clark (Hong and Clark 2001) propose a learning technique that uses stochastic context-free grammars to infer the coarse structure of a page from the abstract strings of HTML tags. The inference technique employs a hill-climbing search over the set of possible grammars which fit with the provided examples, using pre-specified grammar refinement operators and a heuristic based on minimal description length. Then it uses some domain specific rules based on regular expressions to perform a finer extraction of the page, in order to separate the records and the fields within the records. Although their method does not require labeling the example, the user has to write task specific rules manually for the actual extraction. For non-technical users, writing such rules could be a problem. Due to its manual specification of task specific rules to separate the fields, we could classify this method as a manual approach. However if the emphasis is on the document block extraction, then this method can be viewed as an automatic approach.

IEPAD (Chang and Lui 2001) is a system that automatically discovers extraction rules to identify record boundaries within the web pages. Given an HTML page, the system will abstract it as a sequence of tokens. To discover the repeated pattern in the input string, a so-called PAT tree is used. In order to use the PAT tree, the input strings of tokens are represented by several fixed length binary codes. PAT trees are used for exact matching. To allow an inexact match, a technique for multiple string alignment is used. The discovered
repeated patterns are then filtered, and a rule composer creates an extraction rule in the form of a regular expression.

ROADRUNNER (Crescenzi, Mecca, and Merialdo 2001) is a system that generates wrappers automatically. The system works by comparing two HTML pages at a time for similarities and dissimilarities. It uses a heuristic that identifies the structure of HTML pages by observing mismatches between these pages. To generate a wrapper, each input page is transformed into a sequence of tokens. Then, it generates an initial wrapper from one of the two pages. ROADRUNNER uses a matching technique, called ACME, that works by first aligning two sequences of tokens, then solving mismatches that arise from these sequences. When there is a mismatch between initial wrapper and token sequences, the wrapper is generalized. The process terminates if the wrapper has solved all encountered mismatches. This system has been successfully applied to, among others, the Okra and BigBook datasets that we used in our experiments. However, it fails to extract data from, among others, the IAF dataset because it cannot handle disjunctive patterns.

6.3.3.2 Tree-based

Hemnani and Bressan (Hemnani and Bressan 2002) propose a tree alignment algorithm that is based on two heuristics for extracting multiple record web documents. The technique utilizes the HTML tree structure of the documents to discover repeated patterns of the record boundaries within a page, using a tree alignment algorithm.

Cohen (Cohen 1999) implements some heuristics that are useful for IE on top of WHIRL. WHIRL (Cohen 2000) is a “soft” logic system that incorporates the notion of textual similarity, which was developed in the information retrieval community. The use of textual similarity in WHIRL is proposed specifically to deal with the problem of integrating and reasoning with heterogeneous information on the Web. Many terms that occur on the Web are not written in a standard way. For example, the similar terms “AT&T” and “ATT Labs” actually denote the same commercial entity. Using approximate matching and reasoning, WHIRL is developed to provide a type of web knowledge base which is able to cope with the integration of such term disparities. The heuristics in (Cohen 1999) are hand-coded and are used to recognize lists and hotlists in HTML pages. In this sense, WHIRL is not a wrapper induction system but rather a logic system that is programmed with heuristics for recognizing various types of structure in HTML documents. Certain complex heuristics implemented in this paper have been shown to perform as well as or better than the learning approach, discussed in (Cohen and Fan 1999), applied on the same data.

Cohen and Fan (Cohen and Fan 1999) develop techniques to learn general, page-independent heuristics which are used for the extraction of simple lists
and simple hotlists. Their methods can also be used to learn page specific wrappers instead of learning the heuristics (in this case we could classify the techniques as using the semi-automatic approach). They claim that these types of pages are the most common types (75%) of wrappers for WHIRL (Cohen 2000). Cohen and Fan use several propositional learning algorithms, such as RIPPER, CART, etc., to learn the heuristics. The heuristics are learned using an approach similar to the one used for learning page specific wrappers. The approach is to classify whether a parse tree node from an HTML document is positive or negative given a set of labeled parse tree nodes as training examples. If the tree node is classified as positive then it should be extracted. This approach is quite different from typical wrapper induction work, in which a wrapper is learned for a specific page format, and where a new training process is needed for every new page format. Because the method learns general and page-independent heuristics, it only needs to be trained once, and it can also be used to extract some other types of web pages. Cohen and Fan also show that a hybrid system which combines the learned page independent extraction heuristics with the conventional wrapper induction approach could achieve a better performance and would require a smaller number of examples to obtain a comparable set of wrappers. The work of Cohen and Fan is quite similar to ours. Our methods employ a tree automaton to accept or reject a tree with one of its nodes replaced by a special label \( x \). In a sense, the tree automaton classifies a certain tree node which is labeled by \( x \) as either positive or negative. The difference is that our method learns a tree automaton as wrapper. Here the tree automaton is not only functioning as a wrapper but is also reflecting the grammar of the set of HTML or XML pages in the examples, because the tree automaton accepts or reject the whole tree. Their method also requires manual specification of the (nineteen) features that will be used by the learning algorithm, while our methods do not need any features for learning.

### 6.4 Other related work

Besides related work that is specifically applied to information extraction, other related work includes work on wrapper maintenance, XML schema inference, query languages for XML documents, and larger field information integration. Below we shall discuss this related work briefly.

#### 6.4.1 Wrapper maintenance

This thesis focuses on the problem of wrapper induction. While the problem of inducing wrappers has attracted a lot of interest, the work on tools for wrapper maintenance has received less attention. Wrapper maintenance is important because web sources could change in several ways which would result in exist-
CHAPTER 6. RELATED WORK

ing wrappers failing to extract target fields correctly. Chidlovskii (Chidlovskii 2001a) distinguishes three cases of changes in the pages that can cause wrapper failure: context shift, content shift, and structural shift.

The wrapper maintenance process typically consists of the following steps (not all systems consider all of these steps):

- **Verification**: This step detects changes in the page or target field format. The goal is to determine whether a wrapper is working correctly. After this, the system can either proceed to the recovery step or ask the designer to re-label and retrain the wrapper.

- **Recovery**: In this step, the wrapper tries to resume the extraction after acknowledging initial failure. The aim is to extract as many correct fields as possible. Typically some heuristics are used for locating the correct target field. This step does not necessarily lead to the repair of the wrapper. In the case of an unsuccessful attempt to recover the wrapper, the system might ask the designer to re-label and retrain the wrapper.

- **Repairing / reinduction**: If the wrapper maintenance system considers the recovery step to be successful, it can continue to repair or reinduce the wrapper using the newly identified and recovered target fields. Typically this is done automatically by invoking a semi-automatic wrapper induction algorithm using the newly labeled examples produced in the recovery step.

Kushmerick (Kushmerick 2000b) proposes a solution to the problem of wrapper verification based on a regression testing paradigm. He proposes an algorithm RAPTURE that verifies whether a wrapper has correctly extracted the target fields from a document. In this method, each target field is described by a collection of syntactic features, such as the fraction of numeric characters (digit density), letter density, fraction of < and > characters (HTML density), word count, word length, etc. RAPTURE compares the values of various features from the wrapper’s output with those of the verified features. In this process, RAPTURE calculates the probabilities that the feature values of each extracted attribute agree with the verified mean feature values. Finally, the feature probabilities are combined to produce an overall probability that the wrapper is working correctly.

Chidlovskii (Chidlovskii 2001a) proposes a wrapper maintenance system that builds alternative and redundant views using the features derived from the target field (or extracted content). This alternative view serves two purposes. First, it is used to validate the information extracted by the wrapper. Secondly, when the wrapper fails, the alternative view is used to recover the correct target fields. The wrapper can continue to repair itself, in the case of a successful recovery task, or else it can notify the designer that manual repair is required.
6.4 OTHER RELATED WORK

Chidlovskii's system consists of two algorithms. The first one applies the basic recovery strategy combined with content classifiers, which act as the redundant views. The content classifiers use decision trees which learn from a set of syntactic features (similar to the ones used in RAPTURE), such as string length, density of digits, etc., and also from a set of semantic features, such as proper names, abbreviations, urls, time strings, noun phrases, etc. The second algorithm uses backward wrappers to scan files from the end to the beginning, to improve the accuracy of the IE recovery.

Recently, Lerman, et al. (Lerman, Minton, and Knoblock 2003) proposed a system for wrapper maintenance. Their system consists of two machine learning algorithms, one to verify the wrapper (called DATAPROG), and one to recover the wrapper (called Page Template). DATAPROG learns a statistically significant pattern from a set of token sequences derived from the abstraction of the target fields. The target fields can be abstracted in several ways. For example, the token "Boulevard" belongs to the following types: alphanumeric strings, alphabetic strings, capitalized words, etc. DATAPROG uses a so-called hypothesis testing technique to decide whether a pattern is significant. The algorithm grows a pattern tree incrementally by finding all promising specializations of a pattern and then pruning the less promising specializations among patterns of the same length. Finally, it extracts all significant patterns from the pattern tree, including the shorter patterns that generalize over the longer patterns. These patterns are searched in a page when the wrapper does not work correctly. After the recovery step successfully labels a few pages where the wrapper fails, the Page Template algorithm can be invoked to repair the wrapper and to reduce the wrapper using a wrapper induction algorithm.

Typically wrapper verification algorithms learn the typing of the target fields (or the typing of the content). In case of failure, the system tries to search for the type patterns in a page. This is different from the typical approach used by wrapper induction algorithms which learn delimiter-based (or context-based or landmark-based) rules. Thus, wrapper verification algorithms are generally sensitive to the changes in the target field (content) format. Although changes in the content alone would usually not affect the extraction rules, there is a problem when content changes are combined with context or structural changes. This means if the extraction rules failed to work (because of either context or structural changes) and the format of the target field changes drastically, for instance if the price format changed from 20.00 to $20.00, these verification methods might not work correctly. Systems such as DATAPROG are more sensitive to these changes because they use middle level syntactic abstraction of the strings in the target field. Systems such as RAPTURE or Chidlovskii's are less sensitive to these changes because RAPTURE uses more global features such as the number of words and the density of numeric characters, while Chidlovskii's uses a mixture of global syntactic and semantic features.
6.4.2 Schema / DTD inference

Other work on web structure mining aims to find structural similarities between web pages. This type of web structure mining is known as schema discovery and document type definitions (DTD) inference and is different in nature from our work. Schema discovery and DTD inference mine the frequent or common structures of web pages. In contrast, our work attempts to do more than find the common structure of web documents. We also try to find the pattern of the field to be extracted inside the common document structure. Some of the related work in the area Schema discovery and DTD inference is described below.

There has been some work on XML DTD or schema inference, such as (Ludischer, Papakonstantinou, Velikhov, and Vianu 1999; Papakonstantinou and Vianu 2000) that infers a tight XML view DTD from queries. These works, however, infer DTD for view queries only. A more sophisticated algorithm for extracting DTD from XML is the XTRACT system (Garofalakis, Gionis, Rastogi, Seshadri, and Shim 2000). This system works in several steps using the Minimum Description Length (MDL) principle to find the best DTD among the candidates. However, it infers the DTD content model in isolation because the inference algorithm works on the string. Regular grammar inference has been used to infer the content model of SGML documents in (Ahonen 1996). This method separates the document content models (or the right hand side of the DTD productions) and infers regular grammar for each of them. Thus the work of (Ahonen 1996) is similar in spirit to (Garofalakis, Gionis, Rastogi, Seshadri, and Shim 2000). Another work in a similar spirit is (Fernau 2001), which infers an XML DTD from a transformed XML document in the form of a hierarchical sequence of tags.

While the works mentioned in the previous paragraph learn string languages, the work in (Chidlovskii 2001b) infers extended context-free grammars for modeling DTD using a grammatical inference approach. Roughly speaking, an extended context-free grammar is a context-free grammar where the right hand side of its productions is in the form of regular expressions over states (terminals and non-terminals). (Chidlovskii 2001b) also considers determinism for easy XML validation.

Other work in this area is concerned with building DataGuides or schema (Nestorov, Abiteboul, and Motwani 1997; Grumbach and Mecca 1999; Goldman and Widom 1999). Roughly speaking, a schema or DataGuide is a kind of structural summary of semi-structured data. For practical applications and computational reasons, this summary is often approximated (Abiteboul 1997; Grumbach and Mecca 1999; Goldman and Widom 1999). These works use proprietary algorithms for schema discovery. Some other applications do not deal with the task of finding the global schema but instead attempt to find frequent substructures (sub-schema) in semi-structured data (Wang and Liu 1999;
6.4. OTHER RELATED WORK

Toivonen 1999). (Wang and Liu 1999) use a modified version of algorithm to mine association rules (Agrawal and Srikant 1994), and (Toivonen 1999) uses an upgraded first order logic version of algorithm to mine association rules (Dehaspe and de Raedt 1997).

6.4.3 XPath and query languages

XPath (XPath 1999) is a language used to identify particular parts (nodes) of an XML document. In addition to this primary purpose, it also provides basic functions for the manipulation of strings, numbers and Booleans. XPath views an XML document as a tree. XPath has been used to support the major elements of some XML query languages recommended by the World Wide Web Consortium (W3C), such as XSL Transformation (XSLT 1999) and XQuery (XQuery 2002).

The language used by XPath is based on a kind of regular path expression that operates on tree paths. An XPath expression is a slash-separated list of child element names that describes a path through the XML document. The pattern selects elements that match the path. The evaluation of an XPath expression occurs with respect to a context, which might consist of a node, the position and size of a context, or a function. We will use the simple XML document shown in Figure 6.1 to describe the XPath syntax.

```xml
<bib>
  <book subject='''database'''>
    <title> Data on the Web </title>
    <author> Serge Abiteboul </author>
    <author> Peter Buneman </author>
    <author> Dan Suciu </author>
    <publisher> Morgan Kaufmann </publisher>
    <year> 2000 </year>
  </book>
</bib>

<bib>
  <book subject='''ai'''>
    <title> Artificial Intelligence : A Modern Approach </title>
    <author> Stuart Russell </author>
    <author> Peter Norvig </author>
    <publisher> Prentice Hall </publisher>
    <year> 1995 </year>
  </book>
</bib>

Figure 6.1: An XML document.

The most useful XPath expression is a location path. A location path
selects a set of nodes relative to the context node. Such nodes can be elements, attributes, text, comments, root nodes, or any combination of these. Some examples of XPath expressions are as follows:

1. /bib: selects the root element bib;
2. /bib/book: selects all the book elements of the bib element;
4. /bib/*: selects all the child elements of the bib element;
5. /bib/book[1]: selects the first child of the book elements of the bib element;
6. /bib/book[year>1999]: selects all the book elements of the bib element that are published after the year 1999;
7. //book[@subject="ai"]: selects all the book elements with ai as a subject.

Note that if the path starts with a slash / (or double slash //), it represents an absolute (or relative) path to an element.

A number of query languages have been proposed for semi-structured data and XML by the document and database community (Bonifati and Ceri 2000; XQuery 2002; Florescu, Levy, and Mendelzon 1998; Abiteboul, Buneman, and Suciu 2000). As XML is a relatively new topic, not all these languages have been well studied to date (but see (Bonifati and Ceri 2000; Bex, Maneth, and Neven 2000)) and it remains to be seen which languages and features will become standard. We will review some of these languages and features below.

When schema information is present, information extraction can be done by writing queries in an XML query language. Although there is at the moment no standard, several XML query languages have emerged over the past two years. Some of them, like XML-QL (Deutsch, Fernandez, Florescu, Levy, and Suciu 1999) and Lorel (Abiteboul, Quass, McHugh, Widom, and Wiener 1997), are based on query languages developed for semi-structured data. In brief, they consist of a WHERE and a CONSTRUCT clause. The WHERE clause selects parts of the input document, mainly by means of a pattern with variables, while the CONSTRUCT clause determines how these selected parts should be assembled to form the output. Consider, for instance, the following XML-QL query:

```
WHERE <bib>
  <book>
    <title> $t </title>,
```
6.4. OTHER RELATED WORK

<publisher>Prentice Hall </publisher>
</book>
</bib>
CONSTRUCT <publisher>
  <prenhall-publishes> $t </prenhall-publishes>
</publisher>

The WHERE clause selects all models occurring in a <book> element supplied by Prentice Hall. Here $t is a variable. The CONSTRUCT clause specifies that for each match for $t, an element <publisher> should be created with a subelement <prenhall-publishes>. XML-QL also has more advanced features like tag-variables, subqueries, aggregates, and path expressions.

XSL (http://www.w3.org/Style/XSL) is a template-based language developed by W3C. Initially, the aim of this language was to support easy transformations from XML to HTML. However, recent additions lifted XSL to a full-fledged XML transformation language. Although XSL is definitely not a query language in the usual sense, as it is much too procedural and too difficult to use, it is the only transformation language commercially available. Another language is XML-QL (Ceri, Comai, Damiani, Fraternali, Paraboschi, and Tanca 1999), which is graphical and therefore well-suited for supporting a user-friendly interface. Finally, we mention the language Quilt (Chamberlin, Robie, and Florescu 2000) which is a combination of features from XSL with declarative WHERE and CONSTRUCT constructs.

6.4.4 Information integration

Another topic related to structured IE is the large field on web information integration. Information integration addresses the problem of integrating heterogeneous data on the Web with the purpose of allowing the user to pose queries to these integrated data. There are different ways to integrate the information on the Web. Some of these are presented and described in (Fensel, Knoblock, Kushmerick, and Roussel 2000; Levy, Knoblock, Minton, and Cohen 1998; Wiederhold 1996: Florescu, Levy, and Mendelzon 1998). A typical information integration system consists of:

- a user interface for entering queries,
- a mediator which consists of a query planner and an execution engine, and
- wrappers that transform data from the original sources into a form that can be processed further by the system.

Some examples of information integration systems are: Tsimmis (Chawathe, Garcia-Molina, Hammer, Ireland, Papakonstantinou, Ullman, and Widom 1994),
CHAPTER 6. RELATED WORK

Information Manifold (Levy, Rajaraman, and Ordille 1996), Ariadne (Knoblock, Minton, Ambite, Modi, Ashish, Muslea, Philpot, and Tejada 1998), WHIRL (Cohen 1998; Cohen 2000), and Jedi (Huck, Fankhauser, Aberer, and Neuhold 1998). When building such a web integration system, the following issues arise:

**Data modeling:** An information integration system works on the pre-existing data. Thus, the designer must first develop the mediated or global schema that describes the selected data sources and reveals the data aspects that might be interesting to the user. Along with the global schema, source descriptions are required by the system. The descriptions specify the mapping of global schema to local schemas of data sources. Source descriptions serve as arbitrators in case of contradictory, overlapping or semantic mismatches, and in case of different naming conventions where different names refer to the same object. Thus the system needs expressive and flexible mechanisms to describe the data. Several techniques are proposed for these purposes. The most well known is the work on XML with a shared DTD. Other techniques work with an ontology, for instance the work on RDF (Resource Description Framework). Some others work with a knowledge representation language based on mathematical logic.

**Query reformulation, optimization and execution:** Because the user poses queries to the global schema, an information integration system must reformulate the user queries into source queries. Clearly, as the language for describing the data sources becomes more expressive, the reformulation process becomes more difficult. Next, the system needs a query execution plan, and building it is the task of a query optimizer. The query execution plan specifies the order in which the different operations in the query are to be performed, and selects the algorithm to be used with each operation. The task of the query optimization engine is difficult for the following reasons. Firstly, the amount of data on the Web is huge and autonomous, which means that statistics about the sources, and their reliability, are not known in advance. Secondly, the structure of the data varies greatly, ranging from semi-structured to unstructured data, and sources differ widely in their processing ability. Thirdly, the data on the Web and their structure are constantly changing. Fourthly, the time needed to access the data may vary across the sources and time. After the query execution plan is completed, it is passed to the query execution engine.

**Wrapping the data sources:** A wrapper is a program that reformats the data from the sources into a format that is usable by the query processor of the system. In other words, a wrapper extracts the data into a suitable source schema. An example is when the source is an HTML document. In this case, the wrapper needs to extract a pre-specified set of tuples
from that document. Clearly if the data and the structure of the data changes frequently, the manual development of wrappers is not feasible. As we have mentioned above, the development and the widespread use of XML will help to solve this problem.

To conclude, web integration systems are different from typical heterogeneous database systems. Wrappers, which automatize the mapping of the sources’ data to the source schemas, are important components of a web integration system.

6.5 Structured IE research: the current state

Two important observations can be made about the current state of the structured IE research:

- There is an evolution from “one representation fits all” into multiple representations. The latter offer more flexibility.
- There is no free lunch: manually-built systems are the most expressive.

Some other observations can be found in (Kuhlins and Tredwell 2002) and (Kauchak, Smarr, and Elkan 2002). In the rest of this section, we will elaborate on the two points above.

6.5.1 Towards multiple representation

As previously mentioned in Section 3.2.7, systems that work on semi-structured data are the most flexible. In fact, some IE systems, such as WHISK (Soderland 1999) and BWI (Freitag and Kushnerick 2000), are flexible enough to be tested on the whole range of data from unstructured to structured. These systems have even been shown to be competitive with more specialized systems that work on unstructured or structured texts only. These systems employed single token-oriented representation that is based on a string language. One might be tempted to conclude that a single general representation, which can work on the whole spectrum from unstructured to structured data, is the most desirable option. However, this is actually not the case.

Extracting information from structured documents such as HTML documents has been considered easy for string-based methods. Some researchers suggest that the real challenge is with more unstructured texts. It is true that some string-based methods are able to extract target fields perfectly by learning from just a small number of examples. However, there are extraction tasks in this domain in which string-based methods have problems and perform poorly.
Indeed, extracting information from HTML documents is sometimes difficult because some HTML documents are created manually and the following problems might occur:

- Some HTML documents may not conform to correct HTML document syntax, e.g. they may not include closing tags or may have uneven tag nesting.

- There might be some inconsistencies in the way information is presented in HTML documents. Some examples of these inconsistencies are:
  
  - a column in a table might contain different data types. This makes it difficult to extract a specific data type. For example, the first column might contain both the name and the type of occupation.
  
  - missing values which makes the structure across documents inconsistent.
  
  - the use of tag permutation to represent data, which makes the structure inside and across documents inconsistent. For example,

    <li><i>b>xxx</i></li> and <li><b>yyy</b><i>i</i></li>

Below are several descriptions of difficult extraction problems, taken from (Cohen, Hurst, and Jensen 2002):

- “td nodes such that the sum of the column width of all left-sibling td nodes is 2”,

- “td nodes such that no right-sibling td node contains visible text”, and

- “all items in the second column of a table”.

Most string-based approaches have difficulties when they are used to extract this kind of information. Some tree-based methods that only work by selecting particular nodes without relating them to contextual information also have difficulties when extracting this kind of information.

Our methods yield a structured document, such as an HTML or an XML document, as a tree, and a tree automaton is used to model such a document. As we mentioned previously, the advantage of using a tree automaton for information extraction is that the extraction can be based on some structural context. We can see in the difficult extraction problems (queries) above that target nodes have to be selected based on some structural context. It is proven in (Gottlob and Koch 2002) that the languages represented by tree automata actually have the expressiveness required for web information extraction and are able to capture such contextual dependencies. Thus their results provide a
6.5. STRUCTURED IE RESEARCH: THE CURRENT STATE

strong argument that the use of tree-based representation is very appropriate for tree-structured documents such as HTML and XML documents. However, further empirical work on the use of our tree-based methods (and their extensions) remains to be done.

A better way to perform information extraction is to consider multiple representations for different document types. The idea is to use the representation that is best suited for the document type and task. Cohen, et al. (Cohen, Hurst, and Jensen 2002) argue that the difficult descriptions above may be hard to express using a tree or a token level representation, but may be easier using a visual representation of the rendered page.

Some previous works that have suggested the idea of using multiple representations are (Freitag 1998a) and (Cohen, Hurst, and Jensen 2002). In these two works, multiple representations are constructed using a single knowledge representation language, i.e. first order logic (Mitchell 1997). First order logic allows us to represent different types of texts within a single framework. To illustrate this point, we give two examples below of how string and tree representations in first-order logic could be used to extract the publication year from the XML document given in Figure 6.1.

Example 6.1 (Representation of a string in first order logic). If we view the XML document shown in Figure 6.1 as a sequence of tokens, then one possible rule to extract the publication year is shown in Figure 6.2. The rule means: if there is a sequence of a word “year”, followed by a punctuation ‘>’, and followed by a number that is 4 digits long, then the number is the publication year.

\[
\text{publicationyear}(\text{Pyear}) :\]
\[
\text{word}(\text{year}, \text{P1}), \text{punc}(>, \text{P2}), \text{number}(\text{Pyear}, \text{P3}), \]
\[
\text{length}(\text{Pyear}, 4), \]
\[
\text{next_to}(\text{P1}, \text{P2}), \text{next_to}(\text{P2}, \text{P3}).
\]

Figure 6.2: An example of string-based first-order extraction rule

Example 6.2 (Representation of a tree in first order logic). If we view the XML document shown in Figure 6.1 as a tree, then one possible rule to extract the publication year is shown in Figure 6.3. The rule means: if there is a bib node that has book node as a child and there is a year node that is the child of the book node, then the child of the year node is the publication year.

With first order logic, modifying a specific document representation is relatively simple. In (Freitag 1998a), this can be done by adding some new
publication_year(Pyear) :-
    node(N1,bib), node(N2,book), node(N3,year), node(N4,Pyear),
    edge(N1,N2), edge(N2,N3), edge(N3,N4).

Figure 6.3: An example of tree-based first-order extraction rule

document type specific features, such as the node/1 and edge/3 features shown in Example 6.2. In (Cohen, Hurst, and Jensen 2002), this can be done by writing a new special purpose “builder” for each representation. When a different representation is required, the system only needs some additional or new features that can be used by the same learner. This is different from previous approaches, where each learner commits to one specific view of a document and applies the view to several document types, from structured to unstructured. Using multiple representations, the wrapper can be learned using a document type-specific representation. This is similar to the idea of using the document type-specific representation that is best suited for the document to be processed.

There is reason to believe that the use of multiple representations for different document types, or the use of a document type-specific representation, is better than “one representation fits all”. Previous works that favor using an appropriate representation include the following

• Cohen, et al. (Cohen, Hurst, and Jensen 2002) show that by using multiple representations, their WL2 system, has broader coverage and a faster learning rate than two previous string-based approaches for structured data.

• Our tree automata wrappers can be described as using a document type-specific representation. They exploit the tree structure that is present in structured documents such as HTML and XML documents. These wrappers have been shown to give better performance than some previous string-based methods for structured data.

Actually, Cohen, et al. (Cohen, Hurst, and Jensen 2002) argue that even in structured HTML documents, multiple representations are better than a single representation. They suggest four different views on HTML documents: tree, string, visual, and geometric views.

A current limitation of our tree automata based methods, as opposed to string-based or to multiple representations based methods, is that the extraction is only possible for the whole tree node. String-based or multiple representations based methods can extract a field that is a part of the tree node or even across tree nodes. This type of extraction is usually needed to extract
information from HTML documents. Currently, our tree-base methods are not able to do this kind of extraction without post-processing. Developing extensions of our tree-based methods to cope with this limitation is a direction for future work.

To conclude, although tree automata are expressive enough to extract information from HTML documents, currently using multiple representations would be more desirable. However, this is not necessarily the case for extraction from data-centric documents such as XML documents. This is because the data to be extracted in XML documents typically span a whole node; thus the XML data accommodate tree automata based methods.

6.5.2 No free lunch

In this section, we will illustrate that manually-built IE systems yield the most expressive classes of wrappers, by using four different extraction tasks from a sample document.

For example, we are given a set of XML or HTML documents produced from a certain DTD that contain some interesting information. In these documents, the information is presented using (a kind of) nested list structures. Specifically there might be several items in the information, and every item itself may have a complex substructure. The structure of this kind of document is actually more or less similar to the structure of the (small) Shakespeare dataset that we have considered in our experiments. Suppose our example is in XML and the concrete DTD of our example is shown in Figure 6.4.

```
<!ELEMENT INFO   (ITEM*)>
<!ELEMENT ITEM   (NAME, SUBITEM*)>
<!ELEMENT SUBITEM (NAME, (DATA | MODEL | PRICE)+)>
<!ELEMENT NAME   (#PCDATA)>
<!ELEMENT DATA   (#PCDATA)>
<!ELEMENT MODEL   (#PCDATA)>
<!ELEMENT PRICE   (#PCDATA)>
```

Figure 6.4: The DTD of our example

Now consider the following information extraction tasks:

**Task 1:** Extract all items.

**Task 2:** Extract the name and the price from every subitem.

**Task 3:** Extract the name of the second subitem from every item.
Task 4: Extract all subitems from the fifth until the fiftieth subitem from every item, if the total number of the subitems in each item is more than one hundred and the price of the fiftieth subitem is at least $10.

Solving the first task is quite easy. The learner can learn to detect the occurrences of the tag 'item' in the document. Actually the tag 'item' is a good, if not a perfect, indicator of the occurrence of one item record in the example documents. String-based and tree-based semi-automatic IE systems will be able to extract the items perfectly after learning from a few examples. Even fully automatic IE systems, which typically use heuristics that detect some repeating structures in the document without the need of a training example, will be able to extract all items perfectly.

Solving the second task is more difficult than solving the first task. The learner can learn that the tag 'name' and tag 'price' are good indicators of the name and price fields to be extracted, respectively. One difficulty is that the price field sometimes occurs at the place after the name field directly and sometimes not. String-based and tree-based semi-automatic IE systems will be able to extract the items perfectly after learning from one or several examples. However, this extraction task is difficult to solve using fully automatic IE systems. This is because these systems basically do not know which fields the user really wants to extract. They cannot extract the name and price fields only without extracting other fields such as the data and model fields. Actually, they cannot extract the name and price fields only because they are not told to do so by the user and they do not know what the user wants. This case illustrates the limitation of fully automatic IE systems.

The third task is more difficult to solve than the first and the second tasks. Learning to extract the name of the second subitem from every item is difficult if it is based on the subitem's 'name' tag only. This is because a subitem might occur any number of times in each item. Extracting the name of the second subitem from the first item only might be easy, because the learner can learn to skip one subitem's name from the beginning of the document. However, extracting the name of the second subitem of the other items is difficult. This extraction task is clearly beyond the reach of both string-based semi-automatic approaches and fully automatic IE systems.

The problem takes another appearance if we use the tree structure of the XML document. After seeing a few examples, a learner working with tree structure (e.g. of Figure 6.5) might see that the fields of interest are always the first child of the subitems, which themselves are the third child of the items. This rule, if learned, will allow the fields of interest to be located perfectly. This is an example of an extraction task that is difficult to learn by string-based learners but learnable by tree-based learners.

Of the four tasks mentioned above, the fourth task is the most difficult task to solve. Indeed, both string-based and tree-based learners will have difficulties
when learning a wrapper that is able to extract the fields of interest correctly. Learning to extract just the fifth through the fiftieth subitems is already difficult because the learner might be tempted to generalize the wrapper to extract all subitems. If we add to this some other unrestricted constraints, clearly the problem is very difficult to learn. The task might not be learnable even if the learner has an unlimited number of examples. This extraction task is clearly beyond the reach of both string-based and tree-based learners and of fully automatic IE systems.

There are at least two reasons why this kind of problem is not really suitable for both string-based and tree-based learners. They are as follows.

1. The first reason is that the task is not learnable with simple labeling of the examples, because some domain knowledge is needed. The fourth task requires the knowledge of the total number of items and some comparative functions such as $>$, $\leq$, $=$, etc.

2. The second reason is that even if this task were learnable, it would need a lot of examples. In our example task, the user would have to label at least 51 subitems of interest as training examples. Thus learning this task is not practical from the user's point of view.

If we have a language that is sufficiently expressive to specify the position of a member in the list, calculate the number of members in the list, and handle other conditions, then it would be easier to specify the extraction task manually than to learn the wrapper. It is likely that we would be able to specify this extraction task using the currently available query languages for XML or for semi-structured data. It should be obvious that query languages can also solve the first three tasks.
Actually, there is another way to solve the second, the third and the fourth tasks. One could build a database by extracting all items in the examples, including all elements of every subitem. Then these tasks can be solved by writing SQL queries to this database. In this way, the problem is not pushed so much down to the extraction level. If this is the case then a fully automatic IE system is the best choice. However, there are applications where we need to extract the fields of interest spontaneously or directly from the Web, especially for ad-hoc extraction tasks. In these types of applications, building a database and writing a SQL query are not very practical.

Our discussion above is summarized in Table 6.2.

<table>
<thead>
<tr>
<th>Task</th>
<th>Automatic systems</th>
<th>String-based learners</th>
<th>Tree-based learners</th>
<th>Manual systems</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>2</td>
<td>-</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>3</td>
<td>-</td>
<td>-</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>4</td>
<td>-</td>
<td>-</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 6.2: Ability of systems to carry out the four extraction tasks

These tasks clearly illustrate that manually-built IE systems (such as those using query languages) are the most expressive ones for constructing wrappers that can be devised by the user. Thus the no free lunch principle applies here: the more difficult and ad-hoc the task, the more effort required from the user.

### 6.6 Which structured IE method?

After analyzing some application possibilities and limitations of the different types of wrapper, we might be tempted to ask which one is the most suitable wrapper for a given task. IE is a practical field, thus one of the main concerns from the point of view of the user is whether a certain technique is usable in practice. To determine practicality, we need to compare the characteristics of different types of wrappers. We actually could make a comparison of different types of wrappers based on two aspects:

1. The first aspect is how much user involvement is needed for the extraction task. This aspect includes how complicated it is to build a wrapper, the number of examples (and thus how much labeling) are needed, whether specialized skill is needed, whether the system provides the user with an interface that can simplify the development process, etc.

2. The second aspect is how complex and expressive are the wrappers that can be produced.
In the following, we will analyze some characteristics of structured IE systems. Note that the following analysis is based on general observations about the current state of structured IE systems only. Some exceptions may not fit into our classification.

Fully automatic IE systems have the following typical characteristics. They are built only once, making use of a substantial amount of user knowledge and expertise. When these systems are performing their extraction tasks, they require almost no user knowledge or involvement. Their extraction capabilities fit only a specific type of document structure, for instance column type or list type. Moreover these automatic wrappers have limited expressiveness, in the sense that they cannot be tailored for ad-hoc extraction tasks.

String-based and tree-based learners are quite simple to build because they typically only require the user to label a few examples. For this reason they are called semi-automatic approaches. They are typically built for a specific extraction task. If there is a new extraction task, then the user has to train a new wrapper by giving it new examples. Because these wrappers are trainable, they are quite easy to adapt if the user has a specific extraction task. Thus trainable wrapper systems are more expressive than fully automatic IE systems from the user's point of view. Note also that tree-based learners are more expressive than string-based learners because we consider extraction tasks that are based on tree patterns. Thus tree-based learners are able to solve more complex tasks than string-based learners. However, tree-based learners typically require more training examples (and thus labeling) from the user than string-based learners. On the other hand, string-based learners are more widely applicable than tree-based learners. This is because the fields extracted by string-based wrappers are not limited to tree nodes only, but can as well be inside a node or across several nodes.

Manually-built IE systems (such as query languages) require more user involvement and expertise to build than the two previous types of IE systems. This is because the wrappers have to be specified or crafted manually. As with the wrapper induction systems above, these wrappers are built for a specific extraction task. If there is a new extraction task then the user has to specify a new wrapper. In terms of expressiveness, the wrappers built in this way are typically very expressive. Of course the actual expressiveness depends on the query language used. See for example, (Gottlob and Koch 2002) who compare the expressiveness of some tree-based wrapper languages for HTML and XML.

The characteristics of structured IE systems described above are summarized in Figure 6.6. We divide each axis by four different characteristics, which meanings should be quite clear from our previous discussions above. For example, simple extraction ability corresponds to task 1 in Section 6.5.2 and complex extraction ability corresponds to task 4 in Section 6.5.2.
CHAPTER 6. RELATED WORK

![Diagram](image)

**Figure 6.6: Some characteristics of structured IE systems**

The results of our analysis, which are shown in Figure 6.6, are actually quite similar to the results of (Laender, Ribeiro-Neto, Silva, and Teixeira 2002), who also illustrate the trade-off between the degree of automation and the degree of flexibility. The difference is that we concentrate our analysis on the domain of structured documents while (Laender, Ribeiro-Neto, Silva, and Teixeira 2002) also consider the domain of unstructured and semi-structured texts.

Note that the user involvement axis can also be interpreted as the ease of development or time to develop. Typically, automatic IE systems are very easy and quick to develop because they can be used for new IE tasks directly and require little or no user involvement. String-based learners typically require small numbers of examples and little user involvement, and are relatively easy to develop. Tree-based learners typically require moderate user involvement in the labeling of examples, and are also relatively easy to develop. Manually-built wrappers typically require more user involvement and are more difficult to develop than automatic and semi-automatic wrappers.

If only based on the two aspects above, then an optimal wrapper is one that is sufficiently expressive for the task at hand, and at the same time, requires as little user involvement as possible. According to Figure 6.6, there are no systems that are better than the others in all circumstances. Each type of IE systems is optimal in particular circumstances, given its unique capability and expressiveness. We can compose a rule of thumb to select the most suitable wrapper as follows.
6.7 Conclusion

We have surveyed some related work in the area of IE from structured documents. This survey is intended to give an image of an interdisciplinary and rapidly growing field. Researchers from various disciplines have explored many different techniques and the state-of-the-art is advancing rapidly.

We have concluded that the use of existing methods such as query languages, string-based and tree-based semi-automatic approaches, and automatic methods are complimentary.

In the next and final chapter of this thesis, we will describe some future work whose results might change the way we view the structured IE problem.

- If the extraction task is simple, then the optimal choice is to use an automatic IE system;
- If the extraction task is simple to moderate, then the optimal choice is to use a string-based learner;
- If the extraction task is moderate to complex, then the optimal choice is to use a tree-based learner;
- Otherwise, if the extraction task is complex, then the optimal choice is to use a manually-built system;

As mentioned above, we can not rule out exceptions that might occur. This is because there are other factors that might need to be considered, such as the flexibility to select fields that are not necessarily whole node, the need to maximize the quality of the extraction, the need to minimize the number of examples needed, etc. Note also that we assume that the two aspects discussed above, user involvement and extraction capability, are equally important. In actual fact, one aspect might be more important to a particular user than the other aspect. Some other aspects might even be irrelevant to a particular user.

More importantly, we also assume that the complexity of the extraction task is known beforehand. This might not be the case in practice. Some users might prefer to start with the simplest and fastest method to solve their task. Other users might prefer to begin with more expressive methods, in order to determine an upper boundary for the kind of performance that can be achieved with a certain extraction task.

Thus the choice of the most suitable structured IE system is very relative. The ideal choice is very much dependent on the objectives of the user. For example, if the user objective is to build the wrapper as fast as possible and the user is willing to accept errors in the extraction, then a simpler and less expressive method might be used to develop the wrapper.
Chapter 7

Conclusion

This chapter summarizes our thesis and suggests some further work.

7.1 Summary

The “World Wide Web” (WWW) is a single, huge and distributed repository of information that has been in existence since the early nineties. The accessibility of this vast amount of information, however, leaves much to be desired. Internet users have come to rely on search engines to help them find the information they need, and these search engines are often inaccurate or imprecise. We could improve the intelligence of these search engines by improving IR and IE tasks on the Web with the help of web mining technology.

We have determined previously that structured data, in tree form or graph form, exists in almost every aspect of web mining. Our investigation has also revealed the need for more expressive methods in web mining. Thus, there exist many opportunities for the development of both existing and new methods that can utilize structured data. This conclusion has been the motivation for our work in this thesis: the exploration of the tree automaton as a more expressive method for web information extraction.

Our work has focused on the IE from structured documents on the Web such as HTML and XML documents. This work is different from traditional work in IE, which uses unstructured texts. Another notable difference is that our method does not exploit any linguistic knowledge, while traditional IE work relies on it intensively. Instead, we exploit the structure of the information that we want to extract by applying tree automata as extractors to tree-structured documents. Because manually coding extraction rules is a bottleneck in the development process and is not feasible for a dynamic medium such as the Web, we develop techniques that infer tree automata from labeled examples.
automatically.

Chapter 3 contains two parts. The first part is a taxonomy of the IE domain, which consists of IE from unstructured texts, IE from semi-structured data, and IE from structured data. In each of these categories, we have divided the taxonomy further into manual and (semi-) automatic approaches. This taxonomy is proposed to better situate our research.

In the second part, we give a new review of recent tree automata inference work. Some of the previous work that surveyed tree grammar inference methods is already more than 20 years old. Indeed, work on tree automata inference was quite rare until recently. In contrast, the work of finite automata inference comprises most of the work in grammatical inference. Recently, some new tree automata inference algorithms have been proposed, and we have included these in our review. We have also reviewed three regular grammar inference methods on which our tree-based methods are based.

In Chapter 3 we have described IE and the grammatical inference fields separately and in Chapter 4 we have shown how the structural IE problem can be cast as a tree grammar inference problem. Then, we have proposed a framework for two tree automata-based IE approaches which distinguish them from the string-based IE approach. Using tree automata to learn wrappers for information extraction is a new idea. We have elaborated on this approach and have discussed some practical issues that are involved. Finally, we have proposed several tree automaton inference algorithms for information extraction problems. They are as follows.

The \( k \)-testable tree automaton inference algorithm is originally proposed in (Garcia 1993; Knuttila 1993). This algorithm has been shown to be able to identify in the limit any \( k \)-testable tree language in the strict sense from positive examples only. It is an algorithm for ranked trees, while documents are unranked trees. Thus, we have to convert web documents into ranked binary trees before we can apply this algorithm. Because the structural context, which is important for target field identification, can be far away when the tree is binarized, the algorithm needs a rather large \( k \) for capturing the structural or distinguishing context. Consequently, the generalization tends to be rather low, often resulting in rather poor recall. To overcome this problem, we have proposed two generalizations for the \( k \)-testable algorithm, namely, \( g \)-testable and \( gl \)-testable algorithms. In the \( g \)-testable algorithm, the generalization is parameterized by \( l \). It considers generalizations of states (which are trees) where the state labels at the lowest \( l \) levels are replaced by wildcards. The \( gl \)-testable algorithm considers generalization that uses the partial order between different generalizations to limit the search. Experiments show that these generalizations improve the performance of the induced wrappers.

In Chapter 4 we have also proposed a local unranked tree automaton inference algorithm. Our motivation for this is as follows. As we mentioned above,
the generalizations of the tree automata in the g-testable and gl-testable algorithms are obtained by selectively introducing wild-card labels. This gives some modest improvement in recall but does not really solve the problem of binarization. Our unranked tree automaton induction algorithm gives an actual solution to the problem of binarization. If distance between the relevant context and the target is indeed a main factor determining the ability to learn an appropriate automaton, then an algorithm inducing a wrapper directly from the unranked tree should perform even better than the algorithms to induce ranked tree automata.

In Chapter 5 we have experimented with the k-testable algorithm and two extensions of this algorithm, as well as the local unranked tree automaton inference algorithm. The experiments on difficult benchmark datasets show that the performance of the local unranked algorithm is better than the state-of-the-art string based learners, and also to the ranked tree automaton inference algorithms. In fact, the local unranked algorithm is the only one giving optimal results. We have also shown that the local unranked algorithm is able to perform well in an extraction task on the full Shakespeare dataset, which was unfeasible using ranked tree automaton inference algorithms described in Chapter 4.

On the other data sets, where the tree structure of the document is unimportant, our results are less good than those of some string-based methods. This confirms that it remains important to select a method with an appropriate bias, when inducing wrappers. Additionally, in Chapter 5, we have shown the following results.

Advantages of our approach:

- The performance evaluation of our methods on the benchmark datasets, based on precision and recall, indicates that our tree-automata based methods generally perform comparably or better than the current state-of-the-art string-based methods.

- The analysis of the learnability of our methods, using criteria from the field of computational learning theory, shows that the local unranked tree automaton inference algorithm is able to identify in the limit the class of local unranked tree automata and is PAC learnable.

- The analysis of the algorithm complexity shows that our methods, with the exception of the gl-testable method, are efficient and actually can learn faster than some string-based methods employed on the same datasets.

- Our methods learn tree languages, which are more expressive than the string languages. There are at least two benefits arising from this added
expressiveness. Firstly, our methods are able to capture some structural contexts in a document that can be arbitrarily far away in the string representing the document. Secondly, our methods are more generally applicable to any type of document formatting, as they do not require different classes of wrappers for different document formats.

- Our methods require little user intervention and are easy to apply. The user only needs to label a relatively small number of examples in order to train the extractor, while some other manual preprocessing steps are often needed for other methods to learn the extractor.

Limitations of our approach:

- Our methods only output a whole node, which might appear to limit their applicability. Certainly this could be true for HTML formatted documents. However, for data-centric documents such as XML documents, the applicability is not really limited because the data to be extracted is typically a whole node.

- Our methods usually require more examples to learn a comparable wrapper. This situation typically occurs when the identification of target fields does not require dependencies between nodes in the tree but can rely on a local pattern, i.e. the field to be extracted is always surrounded by specific delimiters. In this case, our tree-based methods need more examples to learn a particular extraction rule than methods that automatically focus on local patterns.

- Our methods are possibly slower during extraction than string-based methods because they have to parse the document tree and have to substitute each node with z when extracting information from the document.

- Our methods only extract a single field, or slot, in one run.

- Our methods only extract on structured documents. Indeed our methods cannot be used for text-based IE, and are not intended for it.

Some possible extensions to deal with these limitations are discussed further in the next section. Despite these disadvantages, our results suggest that our tree-based methods work as well as or better than more generally applicable state-of-the-art string-based IE methods, in difficult structured domains. Our results also suggest that utilizing the tree structure of the documents is worthwhile for difficult structured IE tasks.

In Chapter 6 we have elaborated on the IE taxonomy that we proposed in Chapter 3 and have given a survey of the related structured IE work and some related problems. The survey mainly reviews works that are situated
in our taxonomy. Then, we have analyzed the current state of structured IE research. Especially, we have argued that recent research suggests the most flexible approach is not an approach that uses “one representation fits all”, but an approach that uses multiple representations. Another observation is that there is “no free lunch”, in the sense that manually-built systems are the most expressive. Finally we have analyzed the practical issues of selecting which IE method to use. We have found that different approaches, such as query languages, string- or tree-based semi-automatic systems, and fully automatic systems, are complimentary. They are not really competing against each other. The tree-based semi-automatic approach could be seen as an intermediate approach that bridges the gap between string-based semi-automatic approaches and query language approaches. All together, we hope that this chapter gives a new view of structured IE research.

7.2 Further work

This section suggests some further work that is mainly proposed to address some of the limitations of our methods.

7.2.1 Extraction of parts of the text node

As mentioned before, our method is able to extract, for instance, the whole node “Celestijnenlaan 200A, 3001 Leuven” in Figure 7.1, but is not able to extract, for instance, the substring “3001” that denotes the postal code only.

One way to broaden the applicability of our methods is to perform a two-step extraction. A whole node of the tree can be extracted in a first step while a second step, using other techniques, can post-process the selected information to extract a part of it. A string-based method could be used to extract, for example, only the postal code of the full address in Figure 7.1.

Another possible solution is to view each word or any subsequence of texts
at the leaves as a unique node that is a part of the document tree. In this approach, the tree in Figure 7.1 is extended to a tree that is shown in Figure 7.2.

If we consider the unranked tree approach, an advantage of the first approach is that the parsing is done naturally, and there is no need to decide whether to parse the tree differently for a certain extraction task. A disadvantage is that we need to use different methods separately to learn to extract a part of a node. An advantage of the second approach is that we can use the unranked algorithm directly to learn to extract the postal code. The disadvantages are that the tree has to be parsed differently for different extraction tasks because different granularity levels might be needed to split text nodes, and the number of the nodes in the tree examples increases, which adds some computational burden when learning and extracting.

7.2.2 Multi-slot extraction and extraction efficiency

Our tree automata based method extracts only one field for each run. It is an open problem to adapt it to make multi-slot extraction. It is also an open problem to improve the efficiency of the extraction process, which is currently $O(n^2)$.

In each trial in the extraction process, our method puts a special label $x$ into one of the text nodes and then runs the automaton. This method is similar to the method that classifies whether a text node is positive, or negative and thus should not be extracted.

One idea is to use the tree automaton as a parser, instead of using it as a classifier. In this way, a top-down automaton is used instead of the bottom-up tree automaton that is currently used in our method. This tree automaton works in a way similar to the way the XPath expressions work. Similar ideas for a solution might come from work on pattern matching on trees, such as the work of (Neumann and Seidl 1998; Neven and Schwentick). In this way,
the tree automaton only parses the tree once. This approach might solve both problems. However, this idea is quite tricky and further work is certainly needed to verify its applicability.

7.2.3 Alternative methods

In the previous subsections, we have suggested improving the functionality and applicability of our tree-based wrappers. In this subsection we describe some alternative methods for tree automaton inference. There are two directions for possible future work for alternative methods concerning tree automaton inference.

The first possible direction is to explore some more expressive tree languages, especially the unranked ones. As mentioned previously, our tree automaton inference algorithms are based on local tree languages. Although these languages have some desirable properties for information extraction, they are quite limited in expressiveness. If providing more examples is not a problem to the user, then more expressive tree languages could also be used. There exist some other tree languages, some of which are reviewed in Chapter 3, that are more expressive than local tree languages. Given more examples, a more expressive tree language might perform better in learning difficult extraction rules than local tree languages.

The second direction is to explore the use of other learning methods to infer tree languages. Besides the classical way of inferring tree grammars, some well-known learning methods developed in the field of machine learning can, in principle, be used as alternatives to infer tree automata. In fact, some work in this direction already existed. For example, (Lankhorst ; Keller and Lutz 1997) use genetic algorithms to infer pushdown automata or context-free grammars and (Lawrence, Giles, and Fong 2000) use recurrent neural networks to infer a complex grammar. Another interesting possibility is to use inductive logic programming (ILP) techniques. In (Bostrom 1996; Bernard and de la Higuera 1999), some grammatical inference algorithms have been proposed to induce logic programs. On the other hand, some ILP technique have also been used to learn grammars, e.g. in (Pulman and Cussens 2001; Zelle and Mooney 1993). It seems that the field of grammatical inference (GI) and ILP can benefit from each other. In fact, some papers, e.g. (Adriaans and Haas 1999; Muggleton 1999), have already suggested this idea.
Appendix A

Examples of the induced tree automaton

In this appendix, we show one of the input trees and the output for the $k$-testable and unranked algorithms on the small Shakespeare dataset, which is the smallest dataset in our experiments. In each part, one of the actual trees that was used in the experiment is displayed.

A.1 Inference on ranked tree

The binary tree of one of the labeled documents in the small Shakespeare dataset is as follows:

```
act_left
  -- title_both
  -- cdata_lemf
  -- scene_both
  -- title_both
  -- cdata_lemf
  -- speech_both
  -- speaker_both
  -- cdata_lemf
  -- line_both
  -- cdata_lemf
  -- line_both
  -- cdata_lemf
  -- line_both
```

175
A.1. INFERENCES ON RANKED TREE

Below is the output of the $k$-testable algorithm with $k = 3$. The output consists of final.state(State) and transition(Subtree, State).

Total number of states: 29

```prolog
final_state([node(act_left, [node(title_both, [])])]).
```

```prolog
transition(node(act_left, [node(title_both, [node(cdata_leaf, []), node(scene_both, [])])]), node(act_left, [node(title_both, [])])).
```

```prolog
transition(node(line_both, [node(cdata_leaf, []), node(line_both, [node(cdata_leaf, []), node(line_both, [])])]), node(line_both, [node(cdata_leaf, []), node(line_both, [])])).
```

```prolog
transition(node(line_both, [node(cdata_leaf, []), node(line_both, [node(cdata_leaf, []), node(line_both, [])])]), node(line_both, [node(cdata_leaf, []), node(line_both, [])])).
```
transition(node(line_both, [node(cdata_leaf, []), node(line_left, [node(cdata_leaf, []), node(line_left, [])]))], node(line_both, [node(cdata_leaf, []), node(line_left, [])])).

transition(node(scene_both, [node(title_both, [node(cdata_leaf, []), node(speech_both, [])]), node(scene_both, [node(speech_both, [])]), node(scene_both, [node(speech_both, [])])]), node(scene_both, [node(speech_both, [])]), node(scene_both, [node(speech_both, [])])).

transition(node(scene_both, [node(title_both, [node(cdata_leaf, []), node(speech_both, [])]), node(scene_both, [node(speech_both, [])]), node(scene_both, [node(speech_both, [])])]), node(scene_both, [node(speech_both, [])]), node(scene_both, [node(speech_both, [])])).

transition(node(scene_both, [node(title_both, [node(cdata_leaf, []), node(speech_both, [])]), node(scene_both, [node(speech_both, [])]), node(scene_both, [node(speech_both, [])])]), node(scene_both, [node(speech_both, [])]), node(scene_both, [node(speech_both, [])])).

transition(node(speech_both, [node(cdata_leaf, []), node(line_both, [node(cdata_leaf, []), node(line_both, [])])]), node(speech_both, [node(line_both, [])]), node(speech_both, [node(line_both, [])])).

transition(node(speech_both, [node(cdata_leaf, []), node(line_left, [node(cdata_leaf, []), node(line_left, [])])]), node(speech_both, [node(line_left, [])]), node(speech_both, [node(line_left, [])])).

transition(node(speech_both, [node(cdata_leaf, []), node(line_left, [node(cdata_leaf, []), node(line_left, [])])]), node(speech_both, [node(line_left, [])]), node(speech_both, [node(line_left, [])])).

transition(node(speech_both, [node(cdata_leaf, []), node(line_both, [node(cdata_leaf, []), node(line_both, [])])]), node(speech_both, [node(line_both, [])]), node(speech_both, [node(line_both, [])])).

transition(node(speech_both, [node(cdata_leaf, []), node(line_left, [node(cdata_leaf, []), node(line_left, [])])]), node(speech_both, [node(line_left, [])]), node(speech_both, [node(line_left, [])])).
A.1. INFERENCES ON RANKED TREE

transition(node(speech_both, [node(speaker_both, [node(cdata_leaf, []), node(line_both, [])]), node(speech_both, [node(speaker_both, []), node(speech_left, [])])]), node(speech_both, [node(speaker_both, []), node(speech_left, [])])).

transition(node(speech_both, [node(speaker_both, [node(cdata_leaf, []), node(line_both, [])]), node(speech_left, [node(speaker_both, []), node(speech_left, [])])]), node(speech_both, [node(speaker_both, []), node(speech_left, [])])).

transition(node(speech_both, [node(speaker_both, [node(cdata_leaf, []), node(line_left, [])]), node(speech_both, [node(speaker_both, []), node(speech_both, [])])]), node(speech_both, [node(speaker_both, []), node(speech_both, [])])).

transition(node(speech_both, [node(speaker_both, [node(cdata_leaf, []), node(line_left, [])]), node(speech_left, [node(speaker_both, []), node(speech_left, [])])]), node(speech_both, [node(speaker_both, []), node(speech_left, [])])).

transition(node(speech_both, [node(speaker_both, [node(cdata_leaf, []), node(line_both, [])]), node(speech_left, [node(speaker_both, []), node(speech_left, [])])]), node(speech_both, [node(speaker_both, []), node(speech_left, [])])).

transition(node(speech_left, [node(speaker_both, [node(cdata_leaf, []), node(line_both, [])])]), node(speech_left, [node(speaker_both, [])])).

transition(node(speech_left, [node(speaker_both, [node(cdata_leaf, []), node(line_left, [])])]), node(speech_left, [node(speaker_both, [])])).

transition(node(title_both, [node(cdata_leaf, []), node(scene_both, [node(title_both, []), node(scene_both, [])])]), node(title_both, [node(cdata_leaf, []), node(scene_both, [])])).

transition(node(title_both, [node(cdata_leaf, []), node(speech_both, [node(speaker_both, []), node(speech_both, [])])]), node(title_both, [node(cdata_leaf, []), node(speech_both, [])])).

transition(node(title_both, [node(cdata_leaf, []), node(speech_both, [node(speaker_both, []), node(speech_left, [])])]), node(title_both, [node(cdata_leaf, []), node(speech_both, [])])).
transition(node(title_both, [node(x_leaf, []), node(speech_both, [node(speaker_both, []), node(speech_both, [])]))), node(title_both, [node(x_leaf, []), node(speech_both, [])])).

transition(node(cdata_leaf, []), node(cdata_leaf, [])).

transition(node(line_left, [node(cdata_leaf, [])]), node(line_left, [node(cdata_leaf, [])])).

transition(node(x_leaf, []), node(x_leaf, [])).

A.2 Inference on unranked tree

The unranked tree of one of the labeled documents in the small Shakespeare dataset can be seen below.

```
act
  <- title
  <- cdata
  <- scene
    <- title
      <- cdata
      <- speech
        <- speaker
        <- line
        <- cdata
        <- line
        <- cdata
      <- speech
        <- speaker
        <- line
        <- cdata
        <- line
        <- cdata
    <- speech
      <- speaker
      <- line
      <- cdata
      <- line
      <- cdata
      <- line
      <- cdata
      <- line
      <- cdata
      <- line
      <- cdata
      <- line
      <- cdata
      <- line
      <- cdata
    <- scene
```
A.2. INFEERENCE ON UNRANKED TREE

\[ \text{\textasciitilde} \text{title}_x \]
\[ \text{\textasciitilde} \text{Z} \]
\[ \text{\textasciitilde} \text{speech} \]
\[ \text{\textasciitilde} \text{speaker} \]
\[ \text{\textasciitilde} \text{data} \]
\[ \text{\textasciitilde} \text{line} \]
\[ \text{\textasciitilde} \text{data} \]
\[ \text{\textasciitilde} \text{line} \]
\[ \text{\textasciitilde} \text{data} \]
\[ \text{\textasciitilde} \text{speech} \]
\[ \text{\textasciitilde} \text{speaker} \]
\[ \text{\textasciitilde} \text{data} \]
\[ \text{\textasciitilde} \text{line} \]
\[ \text{\textasciitilde} \text{data} \]
\[ \text{\textasciitilde} \text{speech} \]
\[ \text{\textasciitilde} \text{speaker} \]
\[ \text{\textasciitilde} \text{data} \]
\[ \text{\textasciitilde} \text{line} \]
\[ \text{\textasciitilde} \text{data} \]
\[ \text{\textasciitilde} \text{speech} \]
\[ \text{\textasciitilde} \text{speaker} \]
\[ \text{\textasciitilde} \text{data} \]
\[ \text{\textasciitilde} \text{line} \]
\[ \text{\textasciitilde} \text{data} \]
\[ \text{\textasciitilde} \text{speech} \]
\[ \text{\textasciitilde} \text{speaker} \]
\[ \text{\textasciitilde} \text{data} \]
\[ \text{\textasciitilde} \text{line} \]
\[ \text{\textasciitilde} \text{data} \]
\[ \text{\textasciitilde} \text{speech} \]
\[ \text{\textasciitilde} \text{speaker} \]
\[ \text{\textasciitilde} \text{data} \]
\[ \text{\textasciitilde} \text{line} \]
\[ \text{\textasciitilde} \text{data} \]
\[ \text{\textasciitilde} \text{speech} \]
\[ \text{\textasciitilde} \text{speaker} \]
\[ \text{\textasciitilde} \text{data} \]
\[ \text{\textasciitilde} \text{line} \]
\[ \text{\textasciitilde} \text{data} \]
\[ \text{\textasciitilde} \text{speech} \]
\[ \text{\textasciitilde} \text{speaker} \]
\[ \text{\textasciitilde} \text{data} \]
\[ \text{\textasciitilde} \text{line} \]
\[ \text{\textasciitilde} \text{data} \]
\[ \text{\textasciitilde} \text{speech} \]
\[ \text{\textasciitilde} \text{speaker} \]
\[ \text{\textasciitilde} \text{data} \]
\[ \text{\textasciitilde} \text{line} \]
\[ \text{\textasciitilde} \text{data} \]
\[ \text{\textasciitilde} \text{speech} \]
\[ \text{\textasciitilde} \text{speaker} \]
\[ \text{\textasciitilde} \text{data} \]
\[ \text{\textasciitilde} \text{line} \]
\[ \text{\textasciitilde} \text{data} \]
\[ \text{\textasciitilde} \text{speech} \]
\[ \text{\textasciitilde} \text{speaker} \]
\[ \text{\textasciitilde} \text{data} \]
\[ \text{\textasciitilde} \text{line} \]
\[ \text{\textasciitilde} \text{data} \]

Below is the output of the unranked algorithm with \( k = 3 \). The output consists of \textit{final_state}\\textit{(State)}, \textit{forkstate}\\textit{(Label, Kgrams)} and \textit{state}\\textit{(Label, State)}. In local unranked automata, the states are equal to the labels.

\textbf{Total number of states: 11}

\textit{final_state}\\textit{([node(act_x, [])])}.

\textit{forkstate}\\textit{(node(act_x, []), [[\#, \#, \textit{title}]-[\#, \textit{title}], [\#, \textit{title},]}
scene] [title, scene], [scene, scene, #] - [accept], [scene, scene_x, scene] - [scene_x, scene], [scene_x, scene, #] - [accept], [scene_x, scene, scene] - [scene, scene_x] - [scene, scene_x] - [scene, scene_x]).

forkstate(node(line, []), [**-**]).

forkstate(node(scene, []), [**-**]).

forkstate(node(scene_x, []), [[#, #, title_x] - [# title_x], [# title_x, speech] - [title_x, speech], [speech, speech, #] - [accept], [speech, speech, speech] - [speech, speech], [title_x, speech, speech] - [speech, speech]]).

forkstate(node(speaker, []), [**-**]).

forkstate(node(speech, []), [**-**]).

forkstate(node(title, []), [**-**]).

forkstate(node(title_x, []), [[#, x] - [x], [x, #] - [accept]]).

state(node(cdata, []), node(cdata, [])).

state(node(x, []), node(x, [])).
Bibliography


BIBLIOGRAPHY


BIBLIOGRAPHY


BIBLIOGRAPHY


Lankhorst, M. A genetic algorithm for induction of nondeterministic pushdown automata.


Index

accuracy, 112
alphabet, 14
approach and issues, 75
arity, 20, 21
artificial intelligence, 5
bias, 113
Bigbook dataset, see dataset, Bigbook
cardinality, 14
case frame, 80
characteristic sample, 54
children, see definition, children
Chomsky hierarchy, 25
of tree languages, 26
competing, see definition, competing
computational learning theory, 35
text, see document, content
text-free grammar, 19
derivation trees, 19
derives, 19
language generated, 19
context-free language, 19
conversion to ranked trees, 82
convert Jabel algorithm, 93
covers, see definition, covers
cross-validation, 84
data cleaning, 31
data mining, 6
   process, 7
dataset
Bigbook, 109
full Shakespeare, 116
IAF, 102
Okra, 109
QS, 102
small Shakespeare, 103
definition, 83
children, 84
competing, 93
covers, 84
k-forks, 83
k-root, 83
k-subtrees, 83
level, 84
partition, 84
skeleton, 84
derivation trees, see context-free grammars, derivation trees
deterministic finite automaton, 18
deterministic tree automaton, 20
DFA, see deterministic finite automaton
distinguishing context, 76, 119, 124
document
   content, 29
DOM, 34
DTD, 33
element, 29
HTML, 29
   schema, 32
semi-structured, 27
structured, 26, 27
tag, 29

198
INDEX

three types, 26
unstructured, 27
view, 34
XML, 32
document object model, see document, DOM
document type definitions, see document, DTD
DOM, see document, DOM
DTA, see deterministic tree automaton
DTD
inference, 130
element, see document, element
empty set, 14
empty string, 14
experiment, 101
distinguishing context, 124
error analysis, 112
on larger datasets, 109
on the Syllabus dataset, 122
sample efficiency, 116
with noisy examples, 124
with the ranked tree algorithms, 102
with the unranked tree algorithm, 113

F1 score, 104
finite automaton
acceptance, 17
finite automaton, 17
full Shakespeare dataset, see dataset, full Shakespeare
g-t-testable algorithm, 85
time complexity, 107
genl algorithm, 89
gentree algorithm, 89
gl-testable algorithm, 88
time complexity, 107
global patterns, 113

Gold’s result, 53
grammar
language generated, 56
grammatical inference, 5, 50
applications, 51
characteristics of tree-based methods, 60
definition, 51
examples, 57
from positive examples, 52
quality criteria, 51
regular grammars, 53
tree grammars, 59
upgrades, 60
graph, 15
directed graph, 15
edges, 15
nodes, 15
path, 15
vertices, 15

HTML, see document, HTML
identification in the limit, 37, 55, 57, 94
categorization, 95
local unranked tree languages, 98

IE, 3, 42
applications, 43
as grammatical inference problem, 74
automatically-built structured systems, 145
classical, 45
definition, 42
from semi-structured data, 47
from string-based to tree-based, 72
from structured data, 48
from unstructured text, 45
manually-built structured systems, 139
multiple representation, 155
new view, 171
new view, 137
sample extraction tasks, 159
semi automatically-built structured systems, 142
string-based, 72
structured, 136
structured IE systems, 137
taxonomy, 44
the current state of structured IE research, 155
tree-based, 72
with ranked tree algorithms, 80
with unranked tree algorithm, 91
incremental, 55
information access, 3
information extraction, see IE
information integration, 153
information retrieval, see IR
Internet Address Finder dataset, see dataset, IAF
IR, 3

k-contextual algorithm, 55, 94
k-contextual languages, 55
  theorem, 55
k-forks, see definition, k-forks
k-grams, 55, 95
k-reversible algorithm, 54
k-reversible languages, 54
k-reversible tree automata, 60
k-roots, see definition, k-roots
k-subtrees, see definition, k-subtrees
k-testable string languages, 56
k-testable tree algorithm, 84
time complexity, 107
k-testable tree automata, 63
  useful properties, 63
k-testable tree languages in the strict sense, 84

k-TTLSS, see k-testable tree languages in the strict sense

language, 14
closure, 14
concatenation, 14
equivalence, 24
learnability, 53
prefixes, 14
relationship, 24
superfinite, 52
length, 14
level, see definition, level
local patterns, 113, 123
local tree automaton, 93

machine learning, 5
  supervised, 5
  unsupervised, 5

method
  advantages, 128, 169
  characteristics, 79
  easy, 9
  efficiency, 10, 172
  efficient, 9
  expressive, 9
  learnability, 10
  limitations, 129, 170
  performance, 10

motivation
  general, 1
  specific, 7

multi-slot extraction, 80, 172

noise, 124
  unknown labels, 113, 123
  variability, 113, 123

noisy examples, 124

Okra dataset, see dataset, Okra
other forks, 85, 88
overfitting, 106

PAC, 37
INDEX

definition, 38
efficient learner, 38
sample bound, 38, 119
PAC learnable, 38, 122
PAC model, 118
partition, see definition, partition
positive presentation, 55
positive sample set, 18
precision, 2, 104
prefix, 14
prefix-tree automaton, 18, 54, 55
preprocessing text content, 76
probabilistic tree automata, 64
probably approximately correct, see PAC

query languages, 151
query learning, 37
Quote Server dataset, see dataset, QS
rank, 20, 21
ranked trees, 20
ranked label, 20
recall, 2, 104
regular expressions
language associated, 17
regular expressions, 16
priorities, 17
regular grammar inference, see grammatical inference, regular grammars
regular language, 17
representational issues, 119, 124
reversible context-free grammars, 61
sample efficiency, 117
schema, see document, schema
discovery, 150
sentence, 14
Shakespeare dataset, see dataset, full Shakespeare
single-slot extraction, 80
skeleton, see definition, skeleton
small Shakespeare dataset, see dataset, small Shakespeare
string, 14
concatenation, 14
string automaton, 17
structural information, 7
tag, see document, tag
target forks, 85, 88
thesis
contributions, 11
main goals, 7
organization, 10
tree structure, 8
training and testing process
ranked, 81
unranked, 92
tree, 15
ancestor, 16
child, 16
descendant, 16
internal nodes, 16
leaf, 16
lexicographic order, 16
parent, 16
siblings, 16
tree automata, 20
ranked, 20
ranked vs unranked, 25, 127
unranked, 21
tree grammar inference, see grammatical inference, tree grammars
trends
in IE, 134
in structured IE, 136
trivial partition, 54
unranked label, 21
unranked tree algorithm, 93
incremental version, 95
time complexity, 121
unranked tree automaton, 22
unranked trees, 21
UTA, see unranked tree automaton

Web, 1, 167
   information integration, 153
web mining, 7, 167
whole node, 79
World Wide Web, 1, 167
wrapper, 8, 47, 154
   maintenance, 147
   rule of thumb to select wrapper, 164
wrapper induction, 47

XML, see document, XML
Xpath, 151