2014-10-01

Scalable Detection and Extraction of Data in Lists in OCRed Text for Ontology Population Using Semi-Supervised and Unsupervised Active Wrapper Induction

Thomas L. Packer
Brigham Young University - Provo

Scalable Detection and Extraction of Data in Lists in OCRed Text for Ontology Population Using Semi-Supervised and Unsupervised Active Wrapper Induction

Thomas L. Packer

A dissertation submitted to the faculty of
Brigham Young University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

David W. Embley, Chair
Eric K. Ringger
William A. Barrett
Robert P. Burton
Mark J. Clement

Department of Computer Science

Brigham Young University

October 2014

ABSTRACT

Scalable Detection and Extraction of Data in Lists in OCRed Text
for Ontology Population Using Semi-Supervised and
Unsupervised Active Wrapper Induction

Thomas L. Packer
Department of Computer Science, BYU
Doctor of Philosophy

Lists of records in machine-printed documents contain much useful information. As one example, the thousands of family history books scanned, OCRed, and placed on-line by FamilySearch.org probably contain hundreds of millions of fact assertions about people, places, family relationships, and life events. Data like this cannot be fully utilized until a person or process locates the data in the document text, extracts it, and structures it with respect to an ontology or database schema. Yet, in the family history industry and other industries, data in lists goes largely unused because no known approach adequately addresses all of the costs, challenges, and requirements of a complete end-to-end solution to this task. The diverse information is costly to extract because many kinds of lists appear even within a single document, differing from each other in both structure and content. The lists' records and component data fields are usually not set apart explicitly from the rest of the text, especially in a corpus of OCRed historical documents. OCR errors and the lack of document structure (e.g. HMTL tags) make list content hard to recognize by a software tool developed without a substantial amount of highly specialized, hand-coded knowledge or machine-learning supervision. Making an approach that is not only accurate but also sufficiently scalable in terms of time and space complexity to process a large corpus efficiently is especially challenging.

In this dissertation, we introduce a novel family of scalable approaches to list discovery and ontology population. Its contributions include the following. We introduce the first general-purpose methods of which we are aware for both list detection and wrapper induction for lists in OCRed or other plain text. We formally outline a mapping between in-line labeled text and populated ontologies, effectively reducing the ontology population problem to a sequence labeling problem, opening the door to applying sequence labelers and other common text tools to the goal of populating a richly structured ontology from text. We provide a novel admissible heuristic for inducing regular expression wrappers using an A^* search. We introduce two ways of modeling list-structured text with a hidden Markov model. We present two query strategies for active learning in a list-wrapper-induction setting. Our primary contributions are two complete and scalable wrapper-induction-based solutions to the end-to-end challenge of finding lists, extracting data, and populating an ontology. The first has linear time and space complexity and extracts highly accurate information at a low cost in terms of user involvement. The second has time and space complexity that are linear in the size of the input text and quadratic in the length of an output record and achieves higher F_1-measures for extracted information as a function of supervision cost. We measure the performance of each of these approaches and show that they perform better than strong baselines, including variations of our own approaches and a conditional random field-based approach.

Keywords: information extraction, data, ontology, conceptual modeling, ontology population, grammar induction, wrapper induction, hidden Markov model, HMM, regular expression, regex, OCR, plain text, OCRed text document, list, active learning, unsupervised active learning, document analysis and recognition, historical document.

ACKNOWLEDGMENTS

I thank Lee Jensen and others at Ancestry.com for providing a substantial collection of document page images and associated OCR text to work on, and for other input and financial support. I thank FamilySearch.org for also supplying documents from its scanned book collection and for their encouragement in this project. I thank Stephen W. Liddle and other members of the BYU Data Extraction Research Group for coding the Annotator UI I used to produce most of the ground truth for training and testing my extractors. I thank the BYU CS faculty and secretaries for giving me a chance to earn another degree. I thank my advisor, Dr. Embley, for his financial support, for his time, for his not giving up on my research project even during the rough spots, and for continuing to help me even into his retirement. I learned from Dr. Embley how to make my writing more readable. I thank my parents for their support and encouragement and inspiration throughout the entire process, especially toward the end when the distance was unimaginable and yet the support did not stop. I credit my dad for helping me look toward a "third global approach" while I was still stuck trying to force the second approach to work. I thank my wife for the burden she has carried at home over more years than expected, with less support and attention from me than we both would have wished. I must also acknowledge the guiding and sustaining hand of the Almighty, without whom I never would have been dumb enough to start a PhD so late in the game and without whom I never would have been smart enough to finish.

Table of Contents

List of Figures

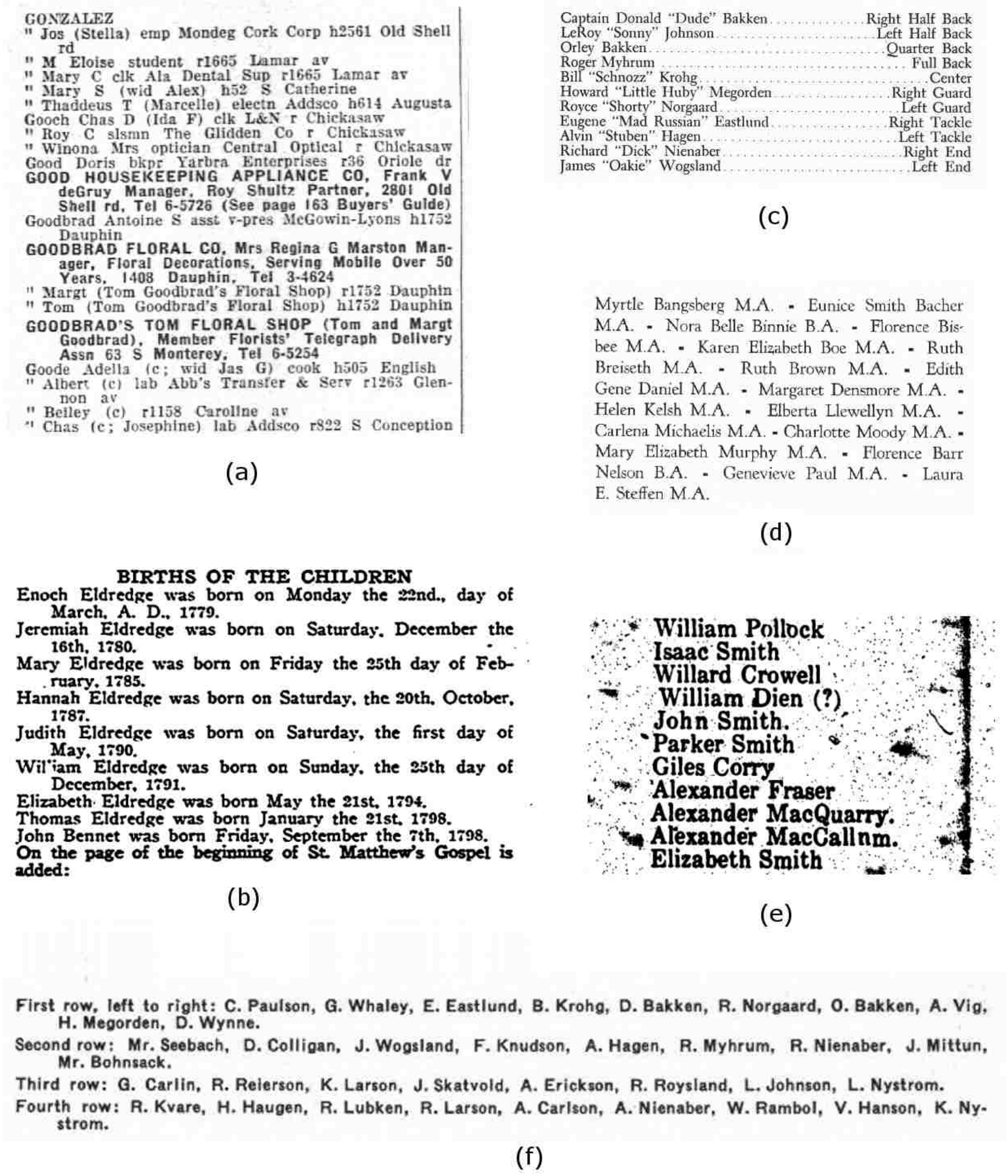
List of Tables

Chapter 1

Introduction

Lists are a data-rich component of many documents. To explain what we mean by “lists” and to clarify the scope of the present research, we first explain our view of lists and their parts using an example and then generalize the example into a definition. The text “`\nJudy Hyde, Jonna Wing, Don Shochacki, Peggy Buday and Janet Martin\n`” is a list containing five records. Each record describes a person and contains two types of fields: the person’s first name and last name, which are attributes of people. (One could also group the two fields into a single full-name field.) The field strings differ from one record to the next. The fields are delimited by a single space, “ ”, and the records are delimited from each other and from surrounding text by members of the following set of strings: “`\n`”, “`, `”, and “` and `”. In general, a list is a sequence of at least two possibly-contiguous records. Each record is a contiguous string of text describing a member of a set of entities, where that set of entities is common to all members of the same list. An entity is an individual object, being, or concept, such as an individual person with a name, height, weight, and other attributes. Each record contains at least one field which is a string representing a property or role of the record’s entity, either directly or indirectly. Fields within a record, and the records themselves, must be separated by strings called delimiters. The corresponding fields in records in the same list share a fixed semantic type or function but do not contain identical text. The record and field delimiters are members of a small, finite set of strings. Other lists appear in Figure 1.1.

Much of the useful information in the records of these lists cannot be fully utilized until this information has been logically connected to a queryable database or ontology schema. A general



GONZALEZ
" Jos (Stella) emp Mondeg Cork Corp h2561 Old Shell rd
" M Eloise student r1665 Lamar av
" Mary C clk Ala Dental Sup r1665 Lamar av
" Mary S (wid Alex) h52 S Catherine
" Thaddeus T (Marcelle) electn Addsco h614 Augusta
Gooch Chas D (Ida F) clk L&N r Chickasaw
" Roy C slsmn The Glidden Co r Chickasaw
" Winona Mrs optician Central Optical r Chickasaw
Good Doris bkpr Yarbra Enterprises r36 Oriole dr
GOOD HOUSEKEEPING APPLIANCE CO, Frank V deGruy Manager, Roy Shultz Partner, 2801 Old Shell rd, Tel 6-5726 (See page 163 Buyers' Guide)
Goodbrad Antoine S asst v-pres McGowin-Lyons h1752 Dauphin
GOODBRAD FLORAL CO, Mrs Regina G Marston Manager, Floral Decorations, Serving Mobile Over 50 Years, 1408 Dauphin, Tel 3-4624
" Margt (Tom Goodbrad's Floral Shop) r1752 Dauphin
" Tom (Tom Goodbrad's Floral Shop) h1752 Dauphin
GOODBRAD'S TOM FLORAL SHOP (Tom and Margt Goodbrad), Member Florists' Telegraph Delivery Assn 63 S Monterey, Tel 6-5254
Goode Adella (c; wid Jas G) cook h505 English
" Albert (c) lab Abb's Transfer & Serv r1263 Glennon av
" Beiley (c) r1158 Caroline av
" Chas (c; Josephine) lab Addsco r822 S Conception

(a)

Captain Donald "Dude" Bakken	Right Half Back
LeRoy "Sonny" Johnson	Left Half Back
Orley Bakken	Quarter Back
Roger Myhrum	Full Back
Bill "Schnozz" Krohg	Center
Howard "Little Huby" Megorden	Right Guard
Royce "Shorty" Norgaard	Left Guard
Eugene "Mad Russian" Eastlund	Right Tackle
Alvin "Stuben" Hagen	Left Tackle
Richard "Dick" Nienaber	Right End
James "Oakie" Wogsland	Left End

(c)

Myrtle Bangsberg M.A. - Eunice Smith Bacher M.A. - Nora Belle Binnie B.A. - Florence Bisbee M.A. - Karen Elizabeth Boe M.A. - Ruth Breiseth M.A. - Ruth Brown M.A. - Edith Gene Daniel M.A. - Margaret Densmore M.A. - Helen Kelsh M.A. - Elberta Llewellyn M.A. - Carlena Michaelis M.A. - Charlotte Moody M.A. - Mary Elizabeth Murphy M.A. - Florence Barr Nelson B.A. - Genevieve Paul M.A. - Laura E. Steffen M.A.

(d)

BIRTHS OF THE CHILDREN

Enoch Eldredge was born on Monday the 22nd., day of March, A. D., 1779.
Jeremiah Eldredge was born on Saturday, December the 16th, 1780.
Mary Eldredge was born on Friday the 25th day of February, 1785.
Hannah Eldredge was born on Saturday, the 20th, October, 1787.
Judith Eldredge was born on Saturday, the first day of May, 1790.
Wil'iam Eldredge was born on Sunday, the 25th day of December, 1791.
Elizabeth Eldredge was born May the 21st, 1794.
Thomas Eldredge was born January the 21st, 1798.
John Bennet was born Friday, September the 7th, 1798.
On the page of the beginning of St. Matthew's Gospel is added:

(b)

William Pollock
Isaac Smith
Willard Crowell
William Dien (?)
John Smith.
Parker Smith
Giles Corry
Alexander Fraser
Alexander MacQuarry.
Alexander MacCallnm.
Elizabeth Smith

(e)

First row, left to right: C. Paulson, G. Whaley, E. Eastlund, B. Krohg, D. Bakken, R. Norgaard, O. Bakken, A. Vig, H. Megorden, D. Wynne.

Second row: Mr. Seebach, D. Colligan, J. Wogsland, F. Knudson, A. Hagen, R. Myhrum, R. Nienaber, J. Mittun, Mr. Bohnsack.

Third row: G. Carlin, R. Reierson, K. Larson, J. Skatvold, A. Erickson, R. Roysland, L. Johnson, L. Nystrom.

Fourth row: R. Kvare, H. Haugen, R. Lubken, R. Larson, A. Carlson, A. Nienaber, W. Rambol, V. Hanson, K. Nystrom.

(f)

Figure 1.1: Example Lists.

method of automatically extracting asserted facts from lists in OCRed documents and inserting them into an ontology would contribute to making a variety of historical and other kinds of knowledge algorithmically searchable, queryable, and linkable. For example, millions of the references to people in family history books are found in lists and include information about birth, marriage, and death events and family relationships. This is the kind of data that could allow patrons of family history websites such as FamilySearch.org and Ancestry.com to easily discover more about their ancestors and complete their family trees. And yet, in family history and other industries, the data in lists goes largely unused because of the cost associated with developing a complete end-to-end process for finding the lists in the document collection, identifying the fields and records within the lists, and populating a database or ontology with the identified data.

Lists are usually not set apart explicitly from the rest of the text, especially in a corpus of historical OCRed documents. In Figure 1.1(b) for example, although “BIRTHS OF THE CHILDREN” (the text before the list begins) is centered and the records in the list are hanging-indented, the OCRed text produced from this image will usually not contain that kind of visual formatting information. Furthermore, we cannot assume that all the lists in that document are indented or marked with the same text. Also, many kinds of lists differ from each other in both layout and content. Therefore, writing individualized code to process all lists is hard and expensive to do right. To work well, a solution to this challenge must be adaptable to variations in document and list format, tolerant of OCR errors, and careful in its selection of human guidance.

As a research area, this challenge deals with semi-structured text and has largely fallen in the cracks between other more traditional and well established fields. Two fields of note are (1) natural language processing that focuses on what some call “unstructured” text and (2) table recognition that focuses on what some call “structured” text including table detection, recognition, and data extraction. Research in these two areas crosses over in limited ways. The research projects most closely related to the present challenge are either too specific, in that they rely on a large amount of specialized, hand-crafted knowledge and rules customized for one kind of list or domain, or they are too general, in that they are not adaptive or customizable by taking advantage of the structure of specific texts to improve accuracy. Machine learning-based approaches to information extraction and ontology population are often more scalable and portable than approaches based on the manual writing of rules and the collection of knowledge resources specific to a single kind of list. However, machine learning approaches often still require a large number of manually selected and annotated examples in order to learn the mapping from facts asserted in text to structured facts asserted in an ontology. This is true because of the amount of variation in semi-structured text which necessitates the hand-labeling of the same kind of field multiple times. We know of no existing work claimed to be a general and complete way to populate an ontology with data from lists, or that streamlines the process end-to-end from text to ontology.

We introduce a way to partially fill the gap between the areas of structured and unstructured information extraction—the gap that contains OCRed and other plain text lists. In research, good or even perfect results are easier to achieve when evaluated against only a single success criterion, ignoring the fact that when the approach is used in practice it will be judged against many other criteria of success simultaneously and will almost certainly be found lacking in an area that was not considered during design and academic testing. To ensure a solid contribution in our chosen challenge, we have worked toward solutions that are acceptable in a wide variety of criteria. In Table 1.1, we summarize our contributions compared to related areas of research and include a list of references for each area. Our goal is to achieve "Good" results over every one of these relevant evaluation criteria. What "Good" means depends on the column, but generally means a level of performance superior to a reasonable baseline. In Table 1.1, cells containing "Very Good" refer to an approach that has either hit an optimal level of performance or the optimal level is within reach with just a little more refinement. "NR" means that the criterion was not reported in the literature, which often means that it was not within the scope of the projects. "Bad" means the approach performs significantly worse in regard to that criterion than most other approaches. "OK" is anything better than than "Bad" and not as good as "Good". The exact meaning of the values and where we draw the line between them may appear subjective, especially considering that most papers do not mention, let a lone numerically measure, their performance against most criteria, and

Table 1.1: Comparison of Approaches over Multiple Evaluation Criteria.

	Input Handling and Project Scope			Process Scalability			IE Output		
	1. OCR/ Noise Tolerance	2. General Text List Detection	3. General Text List Info. Extr.	4. Alg. Time & Space Compl.	5. Knowl. Eng. Cost	6. Annot. Cost	7. Rich Ont. Pop.	8. Prec.	9. Rec.
Related Work									
Grammar Induction [3, 29, 56, 59, 60]	NR	NR	NR	Bad	**Very Good**	**Very Good**	NR	NR	NR
Prob. FSA for IE [12, 35, 44]	**Good**	OK	OK	OK	Bad	OK	OK	**Good**	**Good**
Web Wrapper Induction [13, 15, 18, 20] [22, 24, 32]	NR	NR	OK	**Good**	OK	**Good**	OK	**Good**	**Good**
DAR Systems [7–10, 42]	**Good**	OK	OK	NR	Bad	Bad	OK	**Good**	**Good**
Present Dissertation Work									
List Detection [47]	OK	**Good**	NR	**Very Good**	OK	**Very Good**	NR	NR	NR
Local Regex [48]	**Good**	OK	**Good**	Bad	**Good**	OK	**Good**	**Very Good**	OK
Local HMM [48]	**Good**	OK	**Good**	OK	**Good**	OK	**Good**	**Good**	**Good**
Global Regex [49]	OK	**Good**	**Good**	**Very Good**	**Good**	**Good**	**Good**	**Very Good**	OK
Global HMM [50]	**Good**	**Good**	**Good**	**Good**	**Good**	**Good**	**Good**	**Good**	**Good**

different papers within the same area can have different success profiles. However, we chose cutoffs carefully and consistently while reading representative peer-reviewed papers in a way we think will highlight real differences among the most relevant approaches.

We now briefly explain each criteria.

1. **OCR-error and Noise Tolerance**: the ability to produce accurate output despite errors in the input.
2. **Generality of Text List Detection**: the ability to locate a variety of lists throughout a large or small book containing OCRed or plain text and identify their component records.
3. **Generality of Text List Information Extraction**: the ability to extract data from a variety of lists in OCRed or plain text records.
4. **Algorithm Time and Space Complexity**: The analytical asymptotic worst case "Big O" running time and space complexity of the approach's algorithm as a function of the length of the input text and the size of the discovered record patterns or output label alphabet.
5. **Human Cost of Engineering Knowledge**: the amount of time, effort, or equivalent monetary expense required to produce domain- or genre-specific knowledge resources required to port the system to a new domain or genre, including feature-engineering work and resources.
6. **Human Cost of Annotating Examples**: the amount of time, effort, or equivalent monetary expense required to produce hand-labeled training examples required to port the system to a new domain or genre.
7. **Rich Ontology Population**: the expressiveness of, or number of schema constructs in, the extracted information of the system.
8. **Precision**: the accuracy of the information output by the system.
9. **Recall**: the completeness of the information output by the system.

The work presented in this dissertation introduces and evaluates a novel family of solutions to the challenge of populating an ontology with facts asserted in OCRed lists, including a final approach that achieves good results with respect to the criteria in Table 1.1. The software system embodying our approaches is called ListReader, a wrapper-induction solution for list detection and information extraction that is specialized for lists in OCRed and other plain text documents. ListReader addresses issues arising from the whole end-to-end process, beginning with the discovery of lists in a text document, including the identification of fields and records within the lists, and ending with populating an ontology with identified structured data. The first version of ListReader was called ListDetector because it only detected lists. Subsequent versions of ListReader complete the list reading process by inducing a data-extraction wrapper containing either a regular expression or Hidden Markov Model. ListReader induces each kind of wrapper in both a “local” and a “global” fashion. The local approach reads only a single contiguous list at a time and, within that list, adjusts the wrapper to read each subsequent record before moving on to the next consecutive record. It does not revisit previous records. The global approach associates and reads records of many types simultaneously, wherever they occur in the input text, contiguous or not. This global approach looks at the “big picture” before establishing the details of the wrapper that address individual records.

We now outline the sequence of projects and papers that embody our contributions, as reported in the body of this dissertation, and explain how the papers relate to each other.

Automatic List Detection. In Chapter 2 [47], we present and evaluate the first general-purpose method for detecting lists in OCRed or plain text of which we are aware. This is an unsupervised approach that relies on a collection of cheaply-constructed language resources (e.g., sets of person names) and various functions that filter the labels provided by those language resources and then measure how list-like the resulting pattern of labels is. This work was a first step in the development of our final grammar induction algorithm that also automatically detects lists. Like this first approach, our final approach includes a measure of “pattern area” that is loosely derived from the Minimum Description Length principle (MDL), is simple to understand, and efficient to compute.

As we moved from the challenge of list detection to the challenge of ontology population, we discovered many ways of labeling text automatically in the research literature, but few ways of populating an ontology. To bridge the gap between typical information extraction approaches that annotate text with labels inserted “in-line” into the text stream, we devised a language for in-line labels that we could use to map labeled text to an ontology. The mapping effectively reduces the ontology population problem to a sequence labeling problem. Using this approach, we can use standard machine learning-based information extraction tools to populate a rich variety of ontologically structured predicates by automatically mapping the text labeled by the sequential labeler. The mapping also allows us to train the sequential labeler from hand-annotated examples using standard or novel machine learning techniques, where a user can easily provide the labeled examples by filling in a data-entry form. ListReader translates the data entry form, itself, into the schema of the ontology. We make use of these specially designed in-line labels that map labeled text to ontologies in Chapters 3, 4, and 5.

Regular Expression and Hidden Markov Model Wrappers for Small, Contiguous Lists. In Chapter 3 [48], we present and evaluate the first two general-purpose methods of which we are aware for inducing a wrapper to extract data from lists in OCRed text. These approaches include an active learning approach to inducing a regular expression wrapper and an adaptive supervised approach to inducing a Hidden Markov Model (HMM) wrapper. Both rely on the user to find each list and both use a local approach to learning, meaning they process records in a single contiguous list, one record at a time, in reading order. These approaches allow the induction of a wrapper for a single list from almost nothing more than a single hand-labeled field per field type. Contributions in this paper include a novel admissible heuristic for inducing regular expression wrappers using an A^* search and one of two ways of modeling list structured text with a hidden Markov model.

We discovered a few limitations in our local approaches. Both approaches (regex and HMM) assume that the user will find the lists of interest and label the first record of each list before wrapper induction or active learning begins. Since these approaches assume that a list is a contiguous block of records and that all records in a list are similar in structure, each wrapper induction process can induce a wrapper for only one—possibly small—contiguous block of records at a time, potentially requiring the user to label the same types of fields and records again whenever they appear in another block of records. Also, these approaches rely on the user labeling every field in a record so that ListReader can know which parts of the record (the fields) are variable across records and which parts (the delimiters) should remain more or less constant. Finally, in the case of regex wrapper induction, its approach to handling the large number of possible combinations of fields in each record is to explicitly search over an exponentially-sized hypothesis space using an A* search over possible record variations. Despite a custom admissible search heuristic that we designed for this problem, this regex induction approach is unable to scale up to the search space of the longest records in our corpora. Between the local and the global approaches, we did considerable work trying to push past the limitations of the local approach. We tried two different global approaches before finding the third one which did work well.

The approach we developed next relies on a fundamentally different way of looking at the lists in a book. Instead of trying to read a list like a human would, one record at a time, it looks for all record-like patterns throughout the book simultaneously. It works in a pipeline by clustering lines of text that contain the same classes of characters and words in the same order (e.g., space, digit, upper and lower case letters, punctuation). Then a user can label one instance of each cluster of records. It assumes, however, that records are never longer than one line and that the corresponding fields of two records can never change order. This first global approach is effective when it works, running noticeably faster and requiring less supervision than the local approaches. However, it does not work in all cases, such as when newline-delimited records span two or more lines of text.

We developed a second global approach and discovered that it is also insufficient despite our adding two improvements compared to the first global approach. The first improvement is the addition of a suffix tree data structure which allows ListReader to find patterns efficiently, even ones that spanned line boundaries. Using a suffix tree, ListReader can align records without knowing which characters are record boundaries. The second is that, instead of relying on a predefined set of character or word classes or parsing rules, it attempts to discover any useful parsing rule, customized to the input text, by applying a rule-selection criterion commonly used in unsupervised grammar induction research called the Minimum Description Length principle or MDL. We spent considerable time attempting to implement a working system. During that time, we discovered one reason why the most straightforward formulation of MDL could never work for our application, which resulted in an unpublished, original mathematical proof. We then designed an expanded formula that would account for context in selecting rules, and we believed that this change could be enough to overcome the limitations we faced in applying MDL. We implemented it and tested it, without success. We also began formulating a mathematical proof for why it should work better than the simpler, traditional MDL formula, also without success. It was hard to accept defeat because the MDL approach to list-grammar induction has the potential to be more adaptive to the input text and more complete and cost-effective in discovering list-like patterns without any human guidance. Despite this, we abandoned the core idea in this approach. In doing so, we did not abandon the suffix tree data structure. We combined that idea with the simplicity, low time and space complexity, and effectiveness of our first global approach, to produce something that does work well.

Regular Expression and Hidden Markov Model Wrappers for Large, Discontiguous Lists. In Chapters 4 [49] and 5 [50], we report two variations on the successful "third global approach" to unsupervised list-grammar induction. They both reduce the amount of supervision required to populate an ontology in a complete end-to-end process for discovering lists and populating an ontology that is linear in time and space complexity with respect to the size of the input corpus and therefore scalable. In Chapter 4, we explain how ListReader induces a regular expression wrapper. In addition to being linear in space and time with respect to the size of the input text, the whole process from start to finish is also linear with respect to the size of the record patterns discovered. This is a lower complexity order than any related research of which we are aware and solves the time and space complexity issue we faced in applying our local regex induction approach to longer records. The regex wrapper also extracts information that is highly precise. However, the regex has lower recall, which motivates us to revisit the HMM which is more flexible and less brittle. Finally, in Chapter 5, we explain how ListReader can apply the same grammar induction process as with regex wrappers to train the structure and parameters of an HMM automatically and then to apply unsupervised active learning to complete the mapping from HMM-labeled text to populated ontology. This final approach achieves “good” results for all our evaluation criteria for our complete end-to-end task of discovering lists and populating an ontology, including improved recall compared to the regex. Chapter 6 concludes this dissertation with a discussion of our conclusions, limitations, and future work.

Chapter 2

Lessons Learned in Automatically Detecting Lists in OCRed Historical Documents

Abstract

Lists are often the most data-rich parts of a document collection, but are usually not set apart explicitly from the rest of the text, especially in a corpus of historical OCRed documents. Many kinds of lists differ from each other in both layout and content. Writing individualized code to process all possible types of lists is an expensive challenge. In the present research, we focus on general list detection, the first step in a larger process of general list reading. Our system, ListDetector, automatically locates lists by automatically filtering noisy word labels obtained from cheaply developed dictionaries and regular expressions. In this paper, we start by describing a simple baseline system—the first system we are aware of to address general list detection in plain text or OCRed documents. From there, we present several of the challenges and corresponding solutions that we discovered as we raised the F-measure of ListDetector from 79% to 86.3%. We compute evaluation metrics against a gold standard corpus of OCRed documents in the family history domain that we have manually annotated for the tasks of list detection and structure recogniton. We will continue adding to this corpus and make it publically available for other researchers to use.

2.1 Introduction

Family history, among other kinds of research, depends on the discoverability of information recorded in unstructured and semi-structured text documents. Lists are often the most data-rich parts of a document collection. Before processing a list, e.g., extracting information from the text of the list, it helps to first find and isolate that list automatically from the surrounding text and to infer its structure. However, lists are usually not set apart explicitly from the rest of the text, especially in a corpus of OCRed historical documents. Even if one kind of list is well delimited, often other kinds of lists—even in the same document—are not, or at least not in the same way. Lists differ from each other in both layout and content. One list may be structured like a "bulleted list" of verbose statements delimited by hard returns. Another list may consist of a short sequence of names in a sentence delimited by commas. OCRed lists are especially inconsistent considering some documents contain little or no punctuation because of image noise removal.

We define list recognition as the process of finding and interpreting the structure of lists in text. List recognition provides structural cues to benefit downstream processes—processes that may target either the text inside or outside of the lists. It is analogous to the better-known process of table recognition. Hu et al. [37] decompose the task of table recognition into (1) table detection and (2) table structure recognition. We likewise decompose list recognition into (1) list detection and (2) list structure recognition. In this paper, we focus on general list detection in OCRed documents.

Existing published efforts process specific types of lists in OCRed documents like bibliographies and tables of content. In such circumscribed applications, list detection is a small and straightforward addition to structure recognition. Writing individualized structure recognition code to process all possible types of lists would be much more challenging. As we target the "long tail" challenge in list detection, we prefer a general approach, one that is adaptive to the format, genre, and domain of each page encountered. A solution to general list detection in plain or OCRed text has never been published as far as we are aware. We now enumerate the most closely related research we have found.

Existing work shows how to recognize certain kinds of machine-printed lists using specialized knowledge, geometric table layout recognition techniques, or hand-crafted rules. Le and Thoma [42] and Green [27] use geometric table layout recognition approaches that we believe will not achieve high generality when targeting arbitrary lists without using additional textual and semantic clues. Like Green, Belaïd [8] recognizes tables of content which are similar to lists. His method relies on part-of-speech tagging and specialized rules which we believe make it less general and scalable. The "layout and language" perspective of Hurst and Nasukawa [39] is a general approach to logical document layout recognition using an n-gram language model (language) to regroup physical blocks of text identified using visual clues (layout). They indicate that this approach may help detect some lists. However, since most text in lists is not grammatical, we expect that it will be challenging to adapt Hurst's approach to arbitrary lists. The preceding work focuses on structure recognition which may contain an element of structure detection or categorization. We are not aware of work focusing on list detection in particular.

We also find work related to the present research in the web information extraction community, including the following. Dalvi et al. [20] detect groups of structured records in HTML documents based on patterns in the noisy field labels produced by inexpensive field recognizers (e.g., third-party dictionaries). Groups of structured records could conceivably contain semi-structured lists as we show in this paper, although it is not clear whether they target lists. Gupta and Sarawagi [32] apply a few heuristics to filter out navigational and verbose lists from a candidate pool consisting of all HTML lists in a web crawl. They rely on HTML list tags which help in detecting HTML lists. In their system, starting from all HTML lists in their corpus, to eliminate navigational lists and lists that are too verbose or textual to contain multi-attribute records, they discard a list for any of the following reasons: (1) it has less than 4 records, (2) it has more than 300 records, (3) it has even one record more than 300 bytes long, (4) more than 20% of its records do not have delimiters, or (5) more than 70% of its records are contained inside anchor tags. Chang et al. [15] demonstrate an efficient algorithm for finding repeated substrings in an HTML page and use it to locate groups of structured records. Embley at al. [23] find and segment records in multi-record web pages using a combination of several heuristics looking for the best record delimiter. The web wrapper induction techniques above generally rely on repeated sequences of HTML tags which are not available in OCRed lists.

We are currently designing a system we call ListDetector to be an adaptive, weakly-supervised system for automatically detecting lists in OCRed documents, at a cost lower than typical supervised machine learning techniques allow. In the current implementation described here, ListDetector locates lists automatically after tuning hyper-parameters on a few pages of hand-annotated training data, relying greatly on weak supervision in the form of dictionary and regular expression matching. ListDetector starts processing a page by assigning labels to word tokens that match any of a set of cheaply constructed dictionaries and regular expressions. Given the resulting sequence of noisy labels in a page, ListDetector detects lists by identifying repeating patterns among subsets of labels.

This present work includes the following contributions. We provide the first version of a new public dataset containing images and corresponding OCR output and annotations for the pages in a variety of documents within the family history domain. We describe the data as it relates to the task of list detection in Section 2.2. We present ListDetector (Section 2.3), starting with a simple baseline version, followed by several of the challenges and corresponding solutions we encountered during development. We empirically evaluate ListDetector on the training data along the way. We present final experimental results in terms of a separate set of test pages in Section 2.4. This is the first proposal and evaluation of a general-purpose list detection algorithm we are aware of. We relate a number of additional lessons learned during the development of ListDetector in Section 2.5 and enumerate conclusions and future work in Section 2.6.

2.2 Task and Data

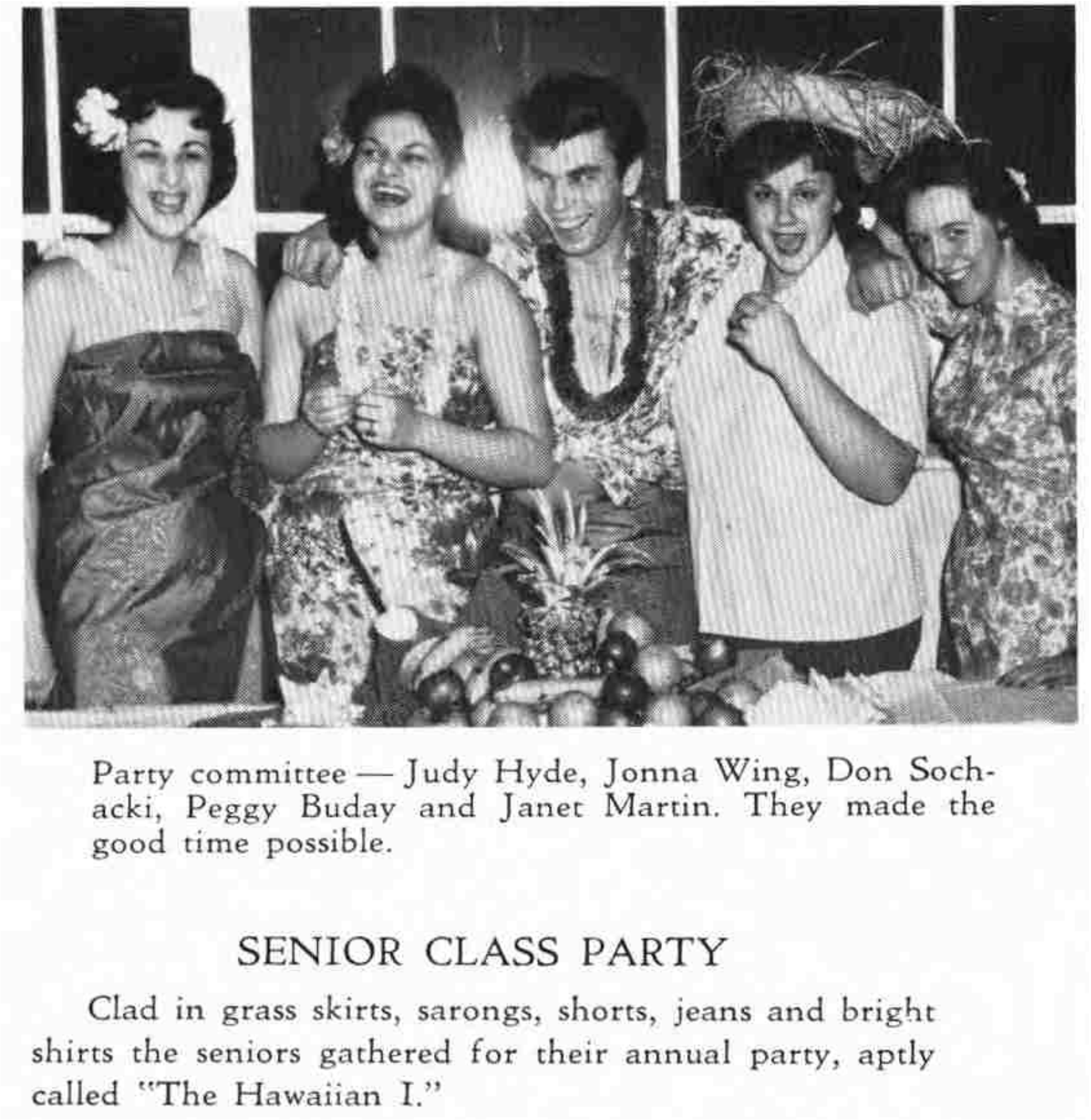
We are currently collecting and annotating a corpus of several kinds of OCRed historical documents which we will provide to the public. For the current project, we manually marked the beginnings and endings of a variety of lists in a sample of pages from two historical newspapers and one high school yearbook. The newspapers are The South Australian Government Gazette, 1867–1884, and The Queensland Government Gazette, 1904–1910. The yearbook is the Bedford High School Abacus, Bedford, Ohio, 1960. We annotated three pages from each of the newspapers (which are all the pages we have available to us) and 36 pages from the yearbook. We then split these pages into a training set consisting of all six pages of newspaper and 20 of the pages of the yearbook, and a test set consisting of the remaining 16 pages of the yearbook. Many of the pages contain no lists, and all the pages contain at least some non-list text, so we have plenty of opportunity to test for false positives. We put all the newspaper pages into the training set because we had already begun development work while visually inspecting these pages before deciding to create a relatively "blind" test set. Figure 2.1 shows part of a yearbook page and Figure 2.2 shows the corresponding OCR text.

Before creating a system to detect lists, we should define what a list is. We are working on a formal definition to propose in future papers. For this paper, we enumerate the criteria that are most relevant to the discussions below.

A list is text consisting of three or more list items called records. Each record consists of one or more fields (substrings), in addition to the record delimiter if present. At least one field in each record must correspond in both semantic role or category and position to a field in some other record. At least one record must contain text that is different from the other records. These criteria must be visible in the OCR text to a person familiar with the corresponding image and the topic (domain) of the document.

Figure 2.3 shows our annotation of the two lists in Figure 2.2.

We experimentally evaluate several versions of ListDetector. When one approach predicts the location of a list, it effectively divides all the word tokens on a page into two categories: list and non-list. Because we compute accuracy metrics from word token counts, it is important to know how we tokenize the text. We split text into word tokens at whitespace boundaries and at transitions between character types. We consider letter, digit, and each type of punctuation character to be a separate character type. Spaces other than newlines are not considered word tokens.

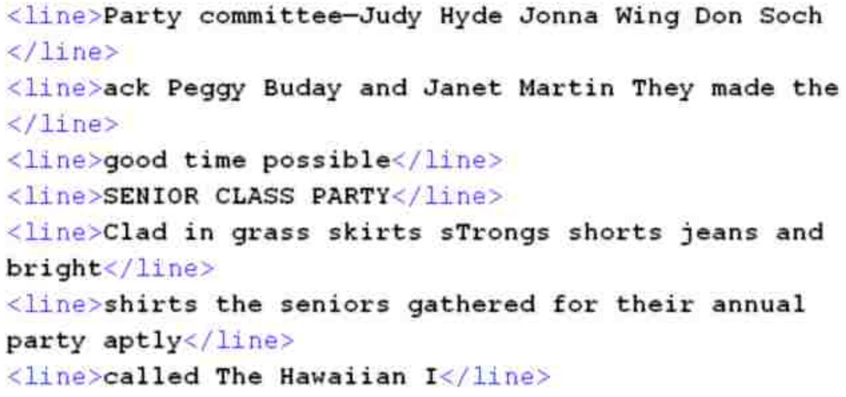


Party committee — Judy Hyde, Jonna Wing, Don Sochacki, Peggy Buday and Janet Martin. They made the good time possible.

SENIOR CLASS PARTY

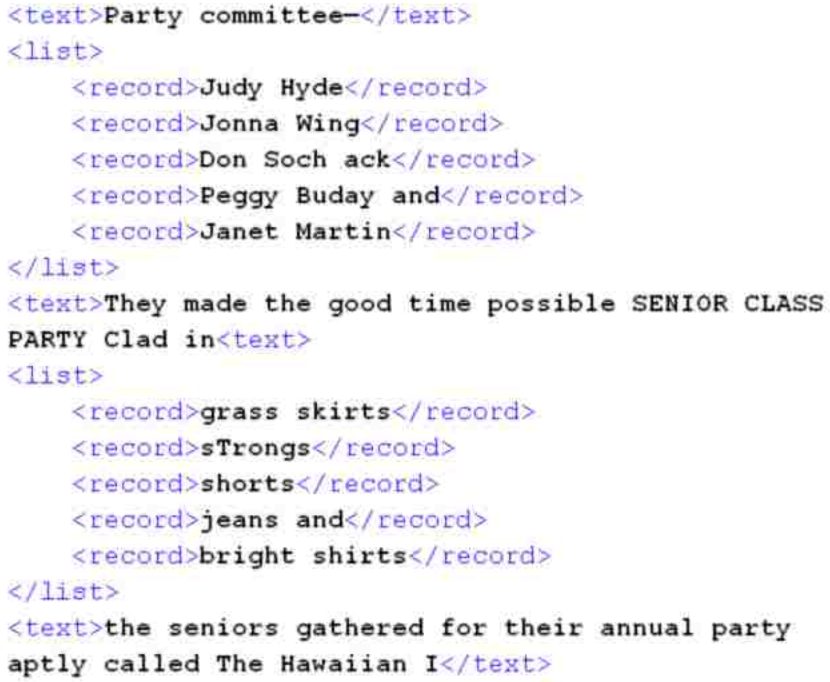
Clad in grass skirts, sarongs, shorts, jeans and bright shirts the seniors gathered for their annual party, aptly called "The Hawaiian I."

Figure 2.1: Part of a Page from a School Yearbook.



```
<line>Party committee—Judy Hyde Jonna Wing Don Soch
</line>
<line>ack Peggy Buday and Janet Martin They made the
</line>
<line>good time possible</line>
<line>SENIOR CLASS PARTY</line>
<line>Clad in grass skirts sTrongs shorts jeans and
bright</line>
<line>shirts the seniors gathered for their annual
party aptly</line>
<line>called The Hawaiian I</line>
```

Figure 2.2: OCR for Image in Figure 2.1.



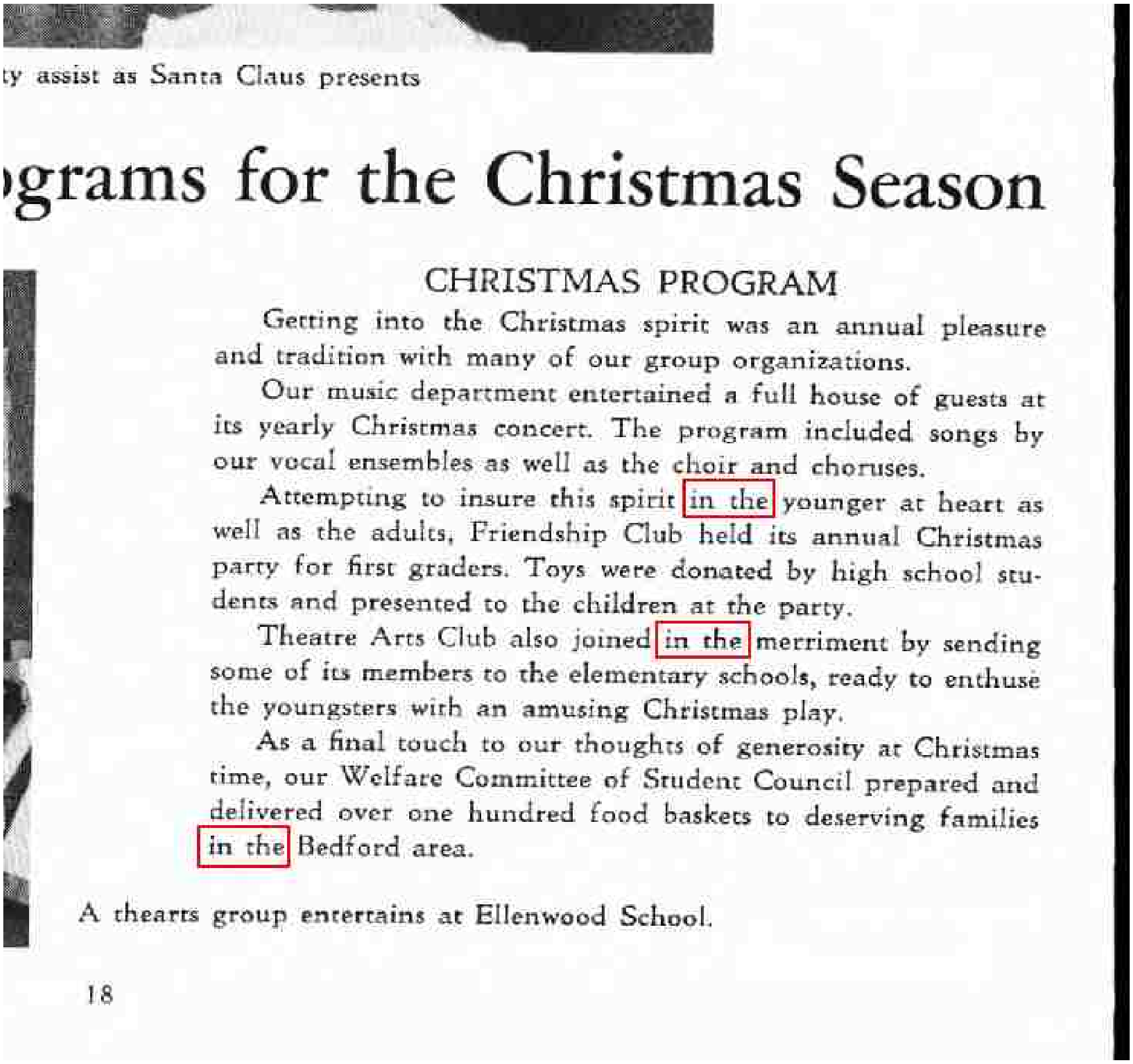
```
<text>Party committee—</text>
<list>
    <record>Judy Hyde</record>
    <record>Jonna Wing</record>
    <record>Don Soch ack</record>
    <record>Peggy Buday and</record>
    <record>Janet Martin</record>
</list>
<text>They made the good time possible SENIOR CLASS
PARTY Clad in<text>
<list>
    <record>grass skirts</record>
    <record>sTrongs</record>
    <record>shorts</record>
    <record>jeans and</record>
    <record>bright shirts</record>
</list>
<text>the seniors gathered for their annual party
aptly called The Hawaiian I</text>
```

Figure 2.3: Expected List Detection and Structure Recognition Output for the Text in Figure 2.2.

We compute precision, recall, and F-measure using standard formulas borrowed from the information retrieval community. *Precision* is the number of words that ListDetector correctly assigns to a list divided by the total number of words it assigns to lists. *Recall* is the number of words ListDetector correctly assigns to a list divided by all words that should have been assigned to a list. *F-measure* (F_1) is the harmonic mean of precision (P) and recall (R):

$$F_1 = \frac{2PR}{P + R}$$

We compute these metrics for each page. We then compute aggregate F-measures over a set of pages in two ways: using macro-averaging and micro-averaging. A macro-average over pages is simply the mean of the metric over those pages. A micro-average is a weighted average—weighted by the number of words in each page. Using both averages gives a more balanced indication of performance. The macro-average emphasizes the most common styles of page and is equivalent to computing F-measure over pages while ignoring the size of the page. The micro-average emphasizes the largest pages and is equivalent to computing the F-measure over all words in the corpus (ignoring



ty assist as Santa Claus presents

grams for the Christmas Season

CHRISTMAS PROGRAM

Getting into the Christmas spirit was an annual pleasure and tradition with many of our group organizations.

Our music department entertained a full house of guests at its yearly Christmas concert. The program included songs by our vocal ensembles as well as the choir and choruses.

Attempting to insure this spirit in the younger at heart as well as the adults, Friendship Club held its annual Christmas party for first graders. Toys were donated by high school students and presented to the children at the party.

Theatre Arts Club also joined in the merriment by sending some of its members to the elementary schools, ready to enthuse the youngsters with an amusing Christmas play.

As a final touch to our thoughts of generosity at Christmas time, our Welfare Committee of Student Council prepared and delivered over one hundred food baskets to deserving families in the Bedford area.

A thearts group entertains at Ellenwood School.

18

Figure 2.4: An Example of a Non-list Prose Text with a False-Positive Pattern Marked by Boxes.

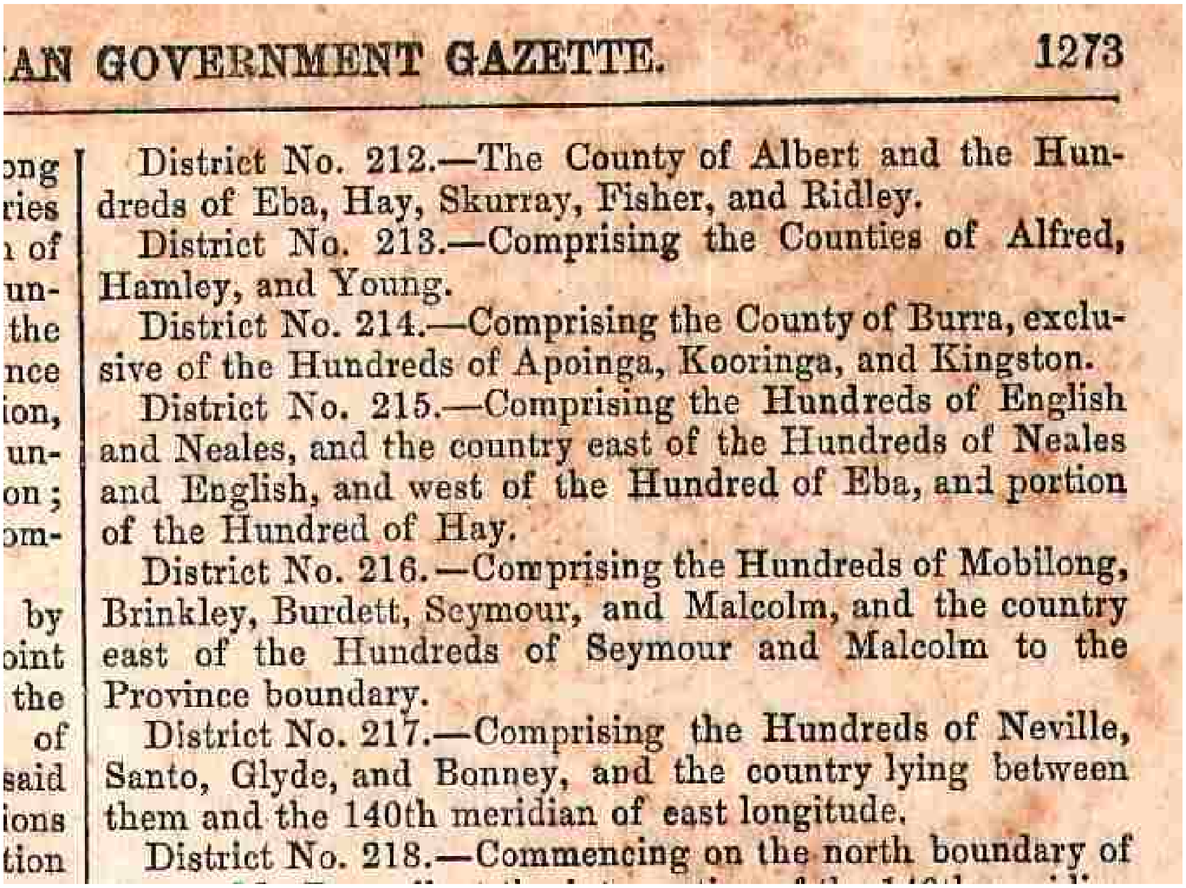
page boundaries). To present a single score by which to compare approaches, we use the average of the macro- and micro-averaged F-measures, and call this the AA score (an average of averages). We believe that an average is a reasonable summary of these two standard ways of computing F-measure in information extraction and information retrieval research.

During development of the below approaches, we tested various parameter settings and chose the ones that maximized performance on the training data, which we report below as we describe each approach. We performed the final evaluation on a separate set of pages to indicate how consistent the relative performance is among approaches, at least on similar types of pages.

2.3 Approaches to List Detection

We designed our first approach to list detection with simplicity, our list definition, and the text in Figure 2.5 in mind. We call it *Simple Literal Pattern Area (SLPA)*. SLPA relies on the assumption that all the records in a list will have some amount of identical text in common. This is a reasonable starting point because it is very easy to implement and many lists do contain identical text, such as "`\nDistrict No.`" in Figure 2.5. SLPA enumerates, scores, and ranks all the literal substrings ("patterns") in an input page. Based on our list definition, SLPA filters out any pattern candidate less than two words long or occurring less than three times. For example, the pattern "`\nDistrict No.`" has a length of four (counting "`\n`", "`District`", "`No`" and "`.`") and a hit count of seven in Figure 2.5, so this pattern is preserved, along with all substrings longer than one word. SLPA scores the remaining patterns using the product of the pattern's length and hit count (the pattern's "area"). Our example pattern has an area score of 28 (4×7). The absolute value is not as important as its relative size compared to alternate patterns. One alternate pattern in our example text, "`-Comprising`", has an area score of 10 (2×5). Finally, SLPA selects the pattern with the highest area score and marks all text between the first and last occurrence of this pattern as a list.

SLPA, as well as all variation reported in this paper, conveniently though incorrectly assume that the first occurrence of a pattern marks the beginning of a list and the last occurrence of a pattern marks the end of a list, which causes ListDetector to discard the last nine tokens in Figure 2.5, namely



AN GOVERNMENT GAZETTE. 1273

ong
ries
n of
un-
the
nce
ion,
un-
on;
om-
by
oint
the
of
said
ions
tion

District No. 212.—The County of Albert and the Hundreds of Eba, Hay, Skurray, Fisher, and Ridley.

District No. 213.—Comprising the Counties of Alfred, Hamley, and Young.

District No. 214.—Comprising the County of Burra, exclusive of the Hundreds of Apoinga, Kooringa, and Kingston.

District No. 215.—Comprising the Hundreds of English and Neales, and the country east of the Hundreds of Neales and English, and west of the Hundred of Eba, and portion of the Hundred of Hay.

District No. 216.—Comprising the Hundreds of Mobilong, Brinkley, Burdett, Seymour, and Malcolm, and the country east of the Hundreds of Seymour and Malcolm to the Province boundary.

District No. 217.—Comprising the Hundreds of Neville, Santo, Glyde, and Bonney, and the country lying between them and the 140th meridian of east longitude.

District No. 218.—Commencing on the north boundary of

Figure 2.5: Part of a Page from a Historical Newspaper.

"218.-Commencing on the north boundary of". The approaches in this paper also recognize only contiguous and constant patterns. For example, if an OCR error had inserted or removed a period in one of the occurrences of "\nDistrict No." in Figure 2.5, that occurrence would not be considered part of the same pattern. We will revisit these two limitations in future research.

The AA score for SLPA on the training data is 51.2%, as we show in the top row of Table 2.1. This score is our starting point, a baseline against which to compare the following developmental changes of ListDetector. One note of caution as we implement changes in response to errors we see in the training data: if an "improvement" is too specific to the training data, then scores computed on the test data may actually decrease at the same time they increase in the training data. This is called over-fitting the training data in the machine learning community.

Approach	Macro-averages			Micro-averages			AA
	Prec.	Rec.	F1	Prec.	Rec.	F1	F1
SLPA	41.0%	57.7%	31.3%	62.3%	82.5%	71.0%	51.2%
BLPA	**77.7%**	56.0%	49.6%	70.9%	82.1%	76.1%	62.8%
RLPA	**77.7%**	61.8%	53.4%	73.1%	93.6%	82.1%	67.7%
PA-NB	65.3%	85.3%	62.7%	74.6%	96.2%	84.0%	73.4%
PA-SD	42.7%	**98.7%**	46.6%	63.3%	**99.6%**	77.4%	62.0%
BPA-SD	55.7%	90.2%	57.3%	66.8%	98.4%	79.6%	68.5%
BPA-SD 2	63.6%	87.6%	62.0%	73.0%	93.8%	82.1%	72.0%
BPA-SD 3	64.5%	89.1%	**63.3%**	**76.9%**	96.9%	**85.7%**	**74.5%**

Table 2.1: Evaluation of Incremental Changes in ListDetector with Respect to the *Training Data*. “AA” is the Average of the Two Kinds of F-measure Averages.

SLPA relies on very lenient constraints on pattern length and hit count to filter spurious lists. As we reviewed the false positives produced by SLPA in the training data (e.g., common bigrams in natural language such as “`in the`” in Figure 2.4), we noticed that stricter constraints would help. Increasing the lower bound on pattern length and hit count independently improved precision, but usually at the expense of recall and F-measure. Raising the lower bound on their product from 6 to 10, however, improved both macro-averaged and micro-averaged F-measure which moves the AA score from 51.2% to 62.8% on the training data. We call this second approach *Bounded Literal Pattern Area (BLPA)*.

BLPA assumes that each page contains at most one list and that this list is spanned by just one pattern. As we inspected the training data, we saw that neither assumption is correct. Our next approach, *Repeated Literal Pattern Area (RLPA)*, addresses this deficiency by marking text spanned by the top three patterns instead of the top one pattern. It ignores patterns of intermediate rank that are covered by higher-ranking patterns. Using multiple patterns raises the AA score in the training data from 62.8% to 67.7%.

RLPA and its predecessors have a glaring deficiency. According to our definition, a list’s records must share one or more fields, but these fields need not contain identical text. Many of the lists in our training data contain no reasonable literal pattern. On the other hand, almost all of them contain semantically related fields that could at least supplement a literal pattern. Consider the three-digit numerals in the records of Figure 2.5 that join the four-word pattern mentioned above (“\nDistrict No.”) with an additional two- or three-word pattern (“.-” or “.-Comprising”). One way we might recognize a repeating pattern of semantically-related fields is to construct named entity recognizers for those fields. However, we wish to avoid this as it is expensive to develop accurate recognizers for the fields in each list in a corpus. Dalvi et al. [20] provide an idea that we adapt to the present setting. We construct cheap, noisy field recognizers in the form of flat dictionaries and regular expressions which have some chance at matching some of the fields in the lists of a given domain. ListDetector will add a label to all the words in a page that match a recognizer.

For the following experiments, we assembled 11 recognizers. Six dictionaries include 8400 given names, 142,000 surnames, 13 person titles, 200 Australian cities, eight Australian states, and 15 religions. Four regular expressions include numerals of between one and four digits in length, upper-case initial letters, with and without a following period, and capitalized words. It is important to note that aside from the category of these 11 recognizers, they were not refined in any way to perform well on this corpus—their contents were taken from external sources. We also include the word itself as a noisy label so that literal patterns are still discoverable. We now want ListDetector to find list-like patterns among this extended vocabulary of symbols just like it did with the vocabulary of the literal text.

Assembling enough field recognizers to ensure that every list in a corpus is sufficiently covered by recognizers means that often ListDetector will assign multiple labels to a given word. In our training data, a single word can often receive four different labels (e.g., *given name*, *surname*, *capitalized word*, and the word itself) and we have even seen words receive five labels (the four just mentioned plus *Australian city*). The primary challenge in using multiple, noisy labels is to construct a single, flat pattern consisting of just one label per token—the “correct” label—and to ensure that the resulting patterns are more indicative of lists than the original, literal text.

To address this challenge, we implemented two ways to select a single label per token. The first method is to train a naïve Bayes classifier to select the most probable label for each token. We do not want to incur the additional cost of hand labeling training data, so we train the classifier on the noisy labels, themselves, as both input features and output classes, training a separate model for each token using all the tokens in a page except for the token it is applied to. To accumulate statistics by which the naïve Bayes parameters are set, we iterate over each noisy label of each token (other than the current token to which the trained model will be applied). The single noisy label of each iteration becomes the value of the hidden class variable to be predicted by the model. Then as features we add all of the noisy labels of the current token as well as its six neighboring tokens. Features are distinguished by an index giving their source token's relative position to the current token within the text. Our justification for using this approach is that, assuming a page contains a list, the words in the list will tend to be in similar contexts (next to neighbors with similar labels) and therefore the label most strongly associated with that context (among those provided by noisy labelers) should be chosen by the model. In other words, we expect that the true labels in a list pattern will produce label co-occurrence counts that dominate any other labels that occur randomly. We call this approach *Pattern Area with Naïve Bayes Label Selector (PA-NB)*. It performs better than RLPA after tuning some hyper-parameters on the training set as described below for the standard deviation approach, increasing the AA score from 67.7% to 73.4%.

Pattern Area with Standard Deviation Label Selector (PA-SD) replaces the naïve Bayes classifier with a simpler heuristic based on a similar justification. If a page holds a list containing at least one type of labeled field, the distance between subsequent instances of that label should not be very random or variable. In other words, statistical measures of spread of the distance between pairs of labels should be low for types of labels that accurately represent fields in lists and should be high for other labels. The statistical measure of spread that we use is the standard deviation. This is inspired by a similar heuristic used by Embley et al. [23] to perform record boundary detection in web pages. Other measures are also possible, such as variance, except that variance gives more weight to outliers which may be problematic with our noisy data which may contain missing labels. Gaps in the label pattern will cause larger values in variance.

To leverage the standard deviation heuristic, ListDetector ranks the available field labels according to standard deviation and selects the best-ranked label for each token. This second heuristic performs poorly at first, lowering the training data AA score from 67.9% to 62.0%. Using 8 patterns instead of 3 patterns per page improves the score only a small amount. Using the noisy labels has naturally improved recall, now that more list content can be used to generate patterns. We must therefore improve precision, which the noisy labels have decreased so much that the F-measure also decreases. Bounding the area score is one way to do this. Changing the lower bound from 10 to 26 increases the AA score to 68.5%. We call this approach *Bounded Pattern Area with Standard Deviation Label Selector (BPA-SD)*.

BPA-SD still produces a lower precision than recall. One reason is, two lists may share the same pattern and yet be separated from each other by some other non-list text which ListDetector classifies as being part of the same list. Another reason is, ListDetector can find patterns on a page without any list. It will select the ones with the lowest standard deviation no matter how high that standard deviation is. To address these two concerns and improve precision, we altered BPA-SD in two ways. In the first (BPA-SD 2), ListDetector removes text from a list between two instances of a pattern whenever they are farther apart than 2.5 times their pattern's standard deviation. In the second (BPA-SD 3), ListDetector will filter out patterns that have a standard deviation larger than 50 word tokens. These two final adjustments raise the AA score to 72.0% and 74.5%, respectively, the highest on the training data in this paper.

2.4 Experimental Evaluation

Table 2.2 shows the results of intermediate and final versions of ListDetector as evaluated against the test data—pages that we did not inspect during development. The raising of the lower bound on pattern area in SLPA that improved precision in the training data without affecting recall did, in fact, affect recall substantially in the test data while at the same time maximizing precision. The

Approach	**Macro-averages**			**Micro-averages**			**AA**
	Prec.	**Rec.**	**F1**	**Prec.**	**Rec.**	**F1**	**F1**
SLPA	55.9%	43.8%	34.5%	74.9%	83.5%	79.0%	56.8%
BLPA	**100.0%**	12.3%	12.4%	**100.0%**	63.2%	77.4%	44.9%
RLPA	**100.0%**	12.3%	12.4%	**100.0%**	63.2%	77.4%	44.9%
PA-NB	79.4%	41.6%	38.3%	86.1%	85.2%	85.6%	62.0%
PA-SD	34.5%	**81.7%**	36.0%	46.6%	**96.5%**	62.8%	49.4%
BPA-SD	63.0%	56.0%	42.4%	72.6%	84.6%	78.1%	60.2%
BPA-SD 2	83.7%	55.5%	**51.1%**	89.3%	83.4%	**86.3%**	**68.7%**
BPA-SD 3	83.7%	55.5%	**51.1%**	89.3%	83.4%	**86.3%**	**68.7%**

Table 2.2: Evaluation of Incremental Changes in ListDetector with Respect to the *Test Data*. "AA" is the Average of the two kinds of F-measure Averages.

next change did nothing to help the situation, namely increasing the number of selected patterns per page. The change did improve recall in the training data. We expect that when we have a larger test set, some improvement will again be visible.

2.5 Additional Lessons

Keeping with the theme implied by this paper's title, we would like to pass on a few other lessons learned (or re-learned) as part of this project.

Extra blank lines and horizontal rules between text blocks appear like list patterns to ListDetector. We added them as unique "words" to our OCR text at first, wanting to preserve as much information about a page as we could. We removed them from the OCR before generating the evaluation metrics reported here.

Using a training set that is not representative of the test set can make it difficult or impossible to perform well (assuming no transfer learning). This is similar to changing the target evaluation metric in the middle of development. It is harder to hit a moving target than a stationary one.

It is sometimes unknown when positive changes in the training data will consistently predict positive changes in the test data. We were more confident that improvements in the training data were not accidental if more than one metric improved, such as both macro-averaged and micro-averaged F-measure. We believed that using both metrics as a guide helped us accept enhancements in ListDetector that actually improved performance on the unseen test data.

In early experiments, we noticed that improving precision when the test set already had higher precision than recall was often a bad idea, even if the change improved the over-all F-measure in the training data. Since F-measure stays close to the lower of its two components, this suggests that it is useful to keep track of which component is the limiting component at each point in development. This observation also reiterates the need to ensure that training and test data are representative of each other so precision is less likely the limiting component in one set at the same time that recall is the limiting component in the other.

Also early in development, we noticed that jittery evaluation metrics (metrics that went up and down drastically in comparison to minor changes in hyper-parameter values) was a sign of a bug (a misplaced parenthesis) in one of our most important numerical functions (standard deviation).

2.6 Conclusions and Future Work

In this paper, we have presented the first proposed method of general list detection in free text or OCRed text we are aware of. We have enumerated some of the challenges and possible solutions in this task and have empirically evaluated and compared these solutions. We conclude that, despite the simplicity of the approaches, ListDetector can categorize text as belonging to lists with 86.3% F-measure based on a micro-average over pages in our test corpus. The same approach performs best in terms of macro-averaged F-measure compared to the other approaches we have tested and therefore it also performs best in terms of our summary AA score. The test data also confirms that using either the naïve Bayes classifier or the standard deviation heuristic to select noisy labels is effective at raising the F-measure if used in conjunction with the additional precision-raising constraints. The parameters set on training data are sufficiently robust to produce very similar relative performance among the approaches on related but distinct test data. Since the label-selection heuristics are tolerant of variations and random omissions of the available labels, they are also robust to OCR errors in the same way they are robust to incomplete dictionaries and natural variations in the lengths of records.

Since we are just beginning to investigate list detection, the research described above can still benefit by additional improvements. Our experiments have been sensitive to the small size of both training and test data and to ListDetector's several hyper-parameters. We plan to overcome these challenges in our ongoing research. A representative sample of pages in both training and test sets is important. Future work should therefore include a larger set of pages of OCRed text from a greater variety of historical documents. The annotations will contain information about list location, list structure, and relational field content.

We have also begun designing and implementing improvements and alternate approaches to list detection by integrating the following additional information and techniques into ListDetector:

1. Leveraging visual layout clues like line spacing, tab stops, and font style,
2. Acknowledging OCR errors by relaxing the strictness or contiguousness of our pattern language,
3. Using patterns discovered on one page to help discover similar patterns on another page that may contain fewer records,
4. Performing list structure recognition as a post-processing step to make corrections in the results of the initial list detection step,
5. Bootstrapping field dictionaries via information extraction from other lists in the same document, and
6. Acknowledging that field recognizers with a high hit-count naturally have a low standard deviation, even when they are not part of a list, by using unsupervised statistical language modeling to locate list-like patterns among the noisy labels that are unlikely to occur by chance (i.e. collocation metrics).

We expect that these ideas, especially the last four, will enable ListDetector to become more adaptive to document style and content with no added cost in terms of knowledge engineering or human supervision.

Chapter 3

Cost Effective Ontology Population with Data from Lists in OCRed Historical Documents

Abstract

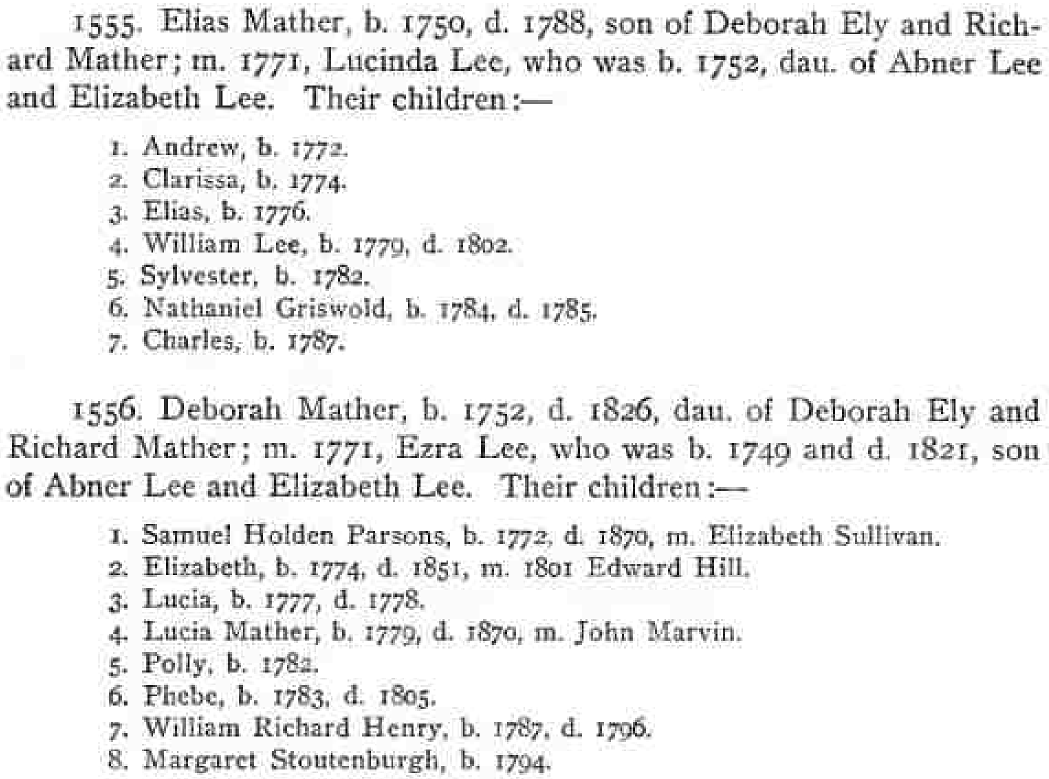
We aim to develop a method of automatically extracting facts from lists in OCRed documents and inserting them into an ontology. Such a method would contribute to making a variety of historical knowledge algorithmically searchable, queryable, and linkable. To work well, such a process must be adaptable to variations in list format, tolerant of OCR errors, and careful in its selection of human guidance. We introduce ListReader, a wrapper-induction solution for information extraction that is specialized for lists in OCRed documents. ListReader can induce either a regular-expression grammar or a Hidden Markov Model. Each can infer list structure and field labels from OCR text. We decrease the cost and improve the accuracy of the induction process using semi-supervised machine learning and active learning, allowing induction of a wrapper from almost a single hand-labeled instance per field per list. After applying an induced wrapper, ListReader automatically maps the labeled text it produces to a rich variety of ontologically structured predicates. We evaluate our implementation on family history books in terms of the typical F-measure and a new metric, "Label Efficiency", which measures both extraction quality and cost in a single number. We show with statistical significance that ListReader reaches values closer to optimal levels than a state-of-the-art statistical sequence labeler.

3.1 Introduction

Family history books and other machine-printed documents present much of their valuable content in data-rich lists. As one example, the 85,000+ family history books scanned, OCRed, and placed on-line by FamilySearch.org are full of lists containing hundreds of millions of fact assertions about people, places, and events. As an example, Figure 3.1 shows lists of the children in two families found on page 154 of *The Ely Ancestry* [6]. These lists make many assertions about family relationships and life events. Our goal is to develop a process to extract the diverse kinds of facts from lists in OCRed documents that is robust to OCR errors and relies on as little human effort as possible. In particular, we are concerned with cheaply extracting rich ontological facts from printed lists in which the up-front cost of extracting information is as low as possible.

To be most useful to downstream search, query, and data-linking applications, the knowledge extracted from text should be expressive and well structured. Ontologies are machine-readable, mathematically specified conceptualizations of a collection of facts. They are expressive enough to provide a framework for storing more of the kinds of assertions found in lists than the typical output of named entity recognition and most other information extraction work. If we could populate user-specified ontologies with predicates representing the facts asserted in OCRed lists, this more expressive and versatile information could better contribute to a number of applications in historical research, database querying, record linkage, automatic construction of family trees, and question answering.

We introduce ListReader, a robust, general, and cost-effective solution to the challenge of extracting diverse types of facts from lists in OCRed documents. ListReader automatically populates a user-defined ontology with assertions found and labeled automatically. A ListReader user constructs an ontology for a list by building a data-entry form in a custom web interface and fills in the form with the information from the first record of a list. ListReader induces a wrapper from a single hand-selected and annotated record and automatically generalizes the wrapper to extract asserted information from the remaining records of the list. Only when ListReader encounters a new field in a later record should it ask the user to update the form to accommodate the new field and



1555. Elias Mather, b. 1750, d. 1788, son of Deborah Ely and Richard Mather; m. 1771, Lucinda Lee, who was b. 1752, dau. of Abner Lee and Elizabeth Lee. Their children:—

1. Andrew, b. 1772.
2. Clarissa, b. 1774.
3. Elias, b. 1776.
4. William Lee, b. 1779, d. 1802.
5. Sylvester, b. 1782.
6. Nathaniel Griswold, b. 1784, d. 1785.
7. Charles, b. 1787.

1556. Deborah Mather, b. 1752, d. 1826, dau. of Deborah Ely and Richard Mather; m. 1771, Ezra Lee, who was b. 1749 and d. 1821, son of Abner Lee and Elizabeth Lee. Their children:—

1. Samuel Holden Parsons, b. 1772, d. 1870, m. Elizabeth Sullivan.
2. Elizabeth, b. 1774, d. 1851, m. 1801 Edward Hill.
3. Lucia, b. 1777, d. 1778.
4. Lucia Mather, b. 1779, d. 1870, m. John Marvin.
5. Polly, b. 1782.
6. Phebe, b. 1783, d. 1805.
7. William Richard Henry, b. 1787, d. 1796.
8. Margaret Stoutenburgh, b. 1794.

Figure 3.1: Lists in *The Ely Ancestry*, Page 154.

insert the field value to provide additional training data. This is a small amount of effort considering the user may be starting a new knowledge extraction project in a new domain and document genre, with no previously assembled resources. After ListReader has begun inducing grammars and extracting information from a document and as the user has started collecting data-entry forms, the cost of extracting information from additional books can decrease further. In this paper we focus on inducing wrappers from scratch based on a mixture of hand-labeled and unlabeled text, making this a semi-supervised learning setting.

Wrapper induction is the automated process of constructing a model (i.e. a grammar) that can extract data from a source document and present it in a uniform format [41]. Each induced wrapper is specifically designed for one data source among many, making it potentially more accurate than applying a single, general model to all of the data sources.

Most work in wrapper induction is specialized for machine-generated HTML pages, although a few projects target lists in HTML documents [22, 32, 43]. The focus on HTML input is reflected in these works' choices in wrapper formalism, which include the following: sets of left and right field context expressions[5, 41], xpaths[20], finite state automata[43], and conditional random fields [22, 32]. These formalisms generally rely on consistent landmarks that are not available in OCRed lists for three reasons: OCRed list text is less consistently structured than machine-generated HTML pages, OCRed text does not contain HTML tags, and field delimiters and content in OCRed documents may contain OCR and typographical errors.

Though not commonly identified as wrapper induction, certain research in extracting information from OCRed printed lists fits our definition and targets input that is similar to our own. Most of this work limits itself to certain kinds of input lists and in applying induction processes that are less adaptable and scalable than our methods. Belaïd [7, 8] and Besagni, et al. [9, 10] extract records and fields from lists of citations, but rely heavily on hand-crafted knowledge that is specific to bibliographies. Adelberg [1] and Heidorn and Wei [35] target lists in OCRed documents in a general sense. They, however, use supervised wrapper induction that we believe is less adaptive or scalable than our methods when encountering the "long tail" of list formats. They do not evaluate cost in combination with accuracy as we do. Also, the extracted information is limited in ontological expressiveness, which is true of all existing work in wrapper induction of which we are aware.

We make the following contributions. (1) After an overview of ListReader in Section 3.2.1, we establish a formal correspondence among list wrappers, ontologies, data-entry forms, and in-line annotated text (Section 3.2.2). This correspondence provides the data-flow for a process in which a user can easily annotate plain text as training data for wrapper induction and create a new ontology schema. It also enables even simple induced wrappers that produce in-line or sequentially labeled text to extract rich facts from lists and insert them into an expressive ontological structure. This effectively reduces the ontology population problem to a sequence labeling problem. (2) We introduce the discovery of new fields in semi-structured text as a novel application of active learning (Section 3.2.3). (3) We demonstrate the induction of wrappers using two formalisms—regular expressions (Regex) (Section 3.2.4) and Hidden Markov Models (HMM) (Section 3.2.5). We show that each can be adaptive to record structure variations such that only about one human-provided label per field is required. We also show that an ensemble of the two wrappers can improve accuracy. (4) We evaluate extraction accuracy combined with the cost of human effort and show with statistical significance that ListReader reaches values much closer to the optimal level than a general, state-of-the-art information extraction system (Section 3.3). We also separately evaluate how complete and concise ListReader's active-learning queries are, showing that it does well with the 60 randomly-chosen child lists in our evaluation set taken from the *The Ely Ancestry* [6]. (5) We conclude that we can benefit from the ListReader line of research and identify opportunities for future research (Section 3.4).

3.2 ListReader

3.2.1 Overview

ListReader[1] populates an ontology from lists in text as follows:

First, a user selects an OCRed image (e.g., a two-layer PDF file) that contains a list (e.g., Figure 3.1), spots a list and, with ListReader's form interface, constructs a form for the data fields in the first record of the list and fills in the form with text from its first record. For example, supposing the spotted list is the second child list in Figure 3.1, the user would construct the form in Figure 3.2 and fill it in by clicking on the words in the PDF document for each field in the form.

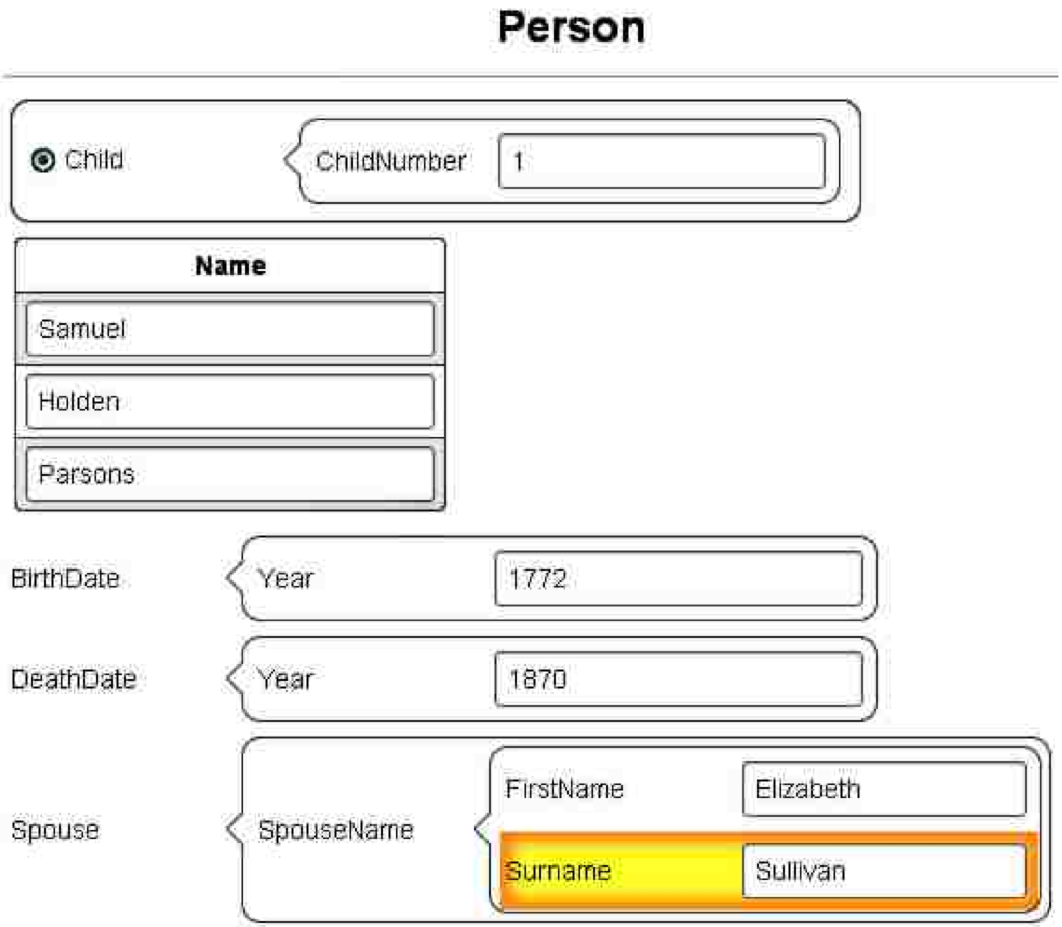
Second, from the empty form, ListReader creates the schema of an ontology (e.g., Figure 3.3).

Third, ListReader uses the information obtained from the filled-in form to label the fields within the text as training data. Figure 3.4 shows the labeled text for our example.

Fourth, ListReader induces a wrapper based on the partially-labeled text, starting with the hand-labeled initial record. It looks for record structure variations in subsequent records of the list, adjusting its wrapper if necessary and asking for user input when it encounters a field not present in the first record.

[1] We have written all the algorithms discussed in this paper from scratch in the Java programming language, using only the libraries typically included in a Java installation, such as the regular expression package java.util.regex.

Finally, the induced wrapper labels the remaining records in the list with labels like those provided in its training data. ListReader translates the labeled text into predicates to insert into the ontology.



Person

● Child
ChildNumber 1

Name
Samuel
Holden
Parsons

BirthDate
Year 1772

DeathDate
Year 1870

Spouse
SpouseName
FirstName Elizabeth
Surname Sullivan

Figure 3.2: Filled in Form for Samuel Holden Parsons Record.

3.2.2 Automatic Mappings

To automate much of ListReader processing, we establish mappings among three types of knowledge representation: (1) HTML forms (e.g., Figure 3.2), (2) ontology structure (e.g., Figure 3.3), and (3) in-line labeled text (e.g., Figure 3.4). The mappings establish a one-to-one correspondence among nested form elements, paths in an ontology graph, and path expressions in text labels. Consider, for example, the birth year “1772” in the child record for Samuel in Figure 3.1. In the

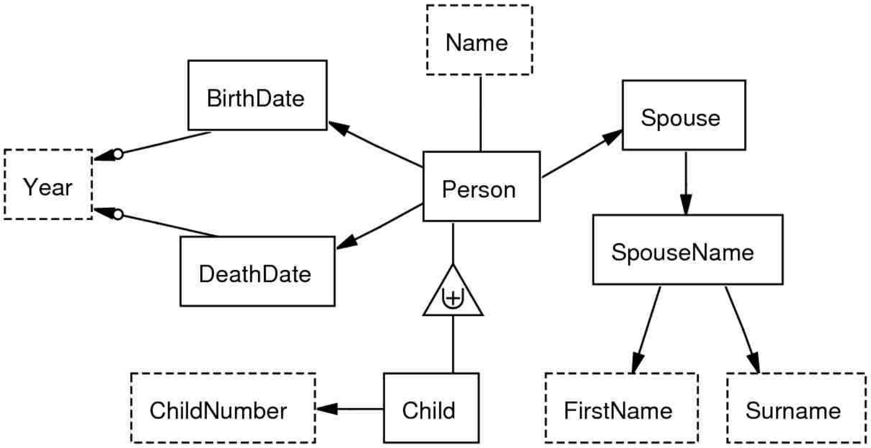


Figure 3.3: Initial List Ontology for Samuel Holden Parsons List.

```
<Child.ChildNumber>1</Child.ChildNumber>.
<Person.Name>Samuel</Person.Name>
<Person.Name>Holden</Person.Name>
<Person.Name>Parsons</Person.Name>
, b. <Person.BirthDate.Year>1772</Person.BirthDate.Year>
, d. <Person.DeathDate.Year>1870</Person.DeathDate.Year>
, m. <Person.Spouse.SpouseName.FirstName>Elizabeth
</Person.Spouse.SpouseName.FirstName>
<Person.Spouse.SpouseName.Surname>Sullivan
</Person.Spouse.SpouseName.Surname>.
```

Figure 3.4: Labeled Samuel Holden Parsons Record.

filled-in HTML form in Figure 3.2, “1772” is in the *Year* field, which is nested under the *BirthDate* field, which is at the top level of nesting under the form title *Person*. This entry in the form generates object and relationship instances in the ontology structure in Figure 3.3: an object identifier to represent Samuel in the object set *Person*, which relates to another object identifier to represent Samuel’s birth date in the object set *BirthDate*, which relates to the text string “1772” in the lexical object set *Year*. In-line labels on fields in text identify these same paths; thus the label for “1772” in Figure 3.4 is *Person.BirthDate.Year*. Because ListReader marks text with labels that have a one-to-one correspondence with the form and ontology for the list, ListReader’s post-processor can fill in the form for the list or, as in our implementation, directly populate the ontology with objects and relationships just as the process that populates the ontology from a manually filled-in form does.

The metaphor of form fill-in for obtaining information is familiar to most users, as is form creation from the basic set of primitives we provide. Our form primitives include the following: a single-entry blank to accept single values (e.g., the birth year "1772" in Figure 3.2), a multiple-entry blank to accept multiple entries (e.g., in the *Name* field in Figure 3.2, the three entries "Samuel", "Holden", and "Parsons"), and radio buttons and check boxes to respectively accept one role or several role designations (e.g., the radio button to designate the *Child* role of *Person* in Figure 3.2). The nesting of form elements provide for relationships among the form elements whose leaf elements are for text objects. Elements with the same name may appear in more than one place in the form, allowing for non-tree-shaped ontologies. The title of the form, *Person* in our example, designates the main object—the object the record describes.

An ontology, rendered as a rooted graph, contains nodes (concepts or sets of objects) and edges (sets of relationships among objects). Nodes may also represent entire concept categorization hierarchies. One object set is designated the *primary object set*. It represents the concept that the list is about, with each record representing a member of that concept. From these ontology primitives ListReader can construct and fill in schemas with the following five points of expressiveness: (1) textual vs. abstract entities (e.g., *Name*("Elias") vs. *Person*(p_1)), (2) n-ary relationships among two or more entities instead of strictly unary or binary relationships (e.g., Husband-married-Wife-in-Year(p_1, p_2, "1771")), (3) ontology graphs with arbitrary path lengths from the root instead of strictly unit-length as in named entity recognition or data slot filling (e.g., *Person.Spouse.SpouseName.Surname*), (4) functional and optional constraints on relationship sets (e.g., A person has one birth year and one death year and a particular year may be one or the other but not necessarily both), and (5) concept categorization hierarchies, including, in particular, role designations (e.g., *Child isa Person*).

In-line labeled text refers to the text annotated with XML-like tags or labels. Each label represents a field as a path in the ontology graph. The path starts at the primary object set, or, in the case when the primary object set is a concept hierarchy, at any concept in the hierarchy. The path ends at the lexical object set for the field whose text is labeled. We denote a graph path through all binary edges by a dot-separated sequence of one or more object-set names. When mapping from labeled text to a populated ontology, field labels denote object and relationship instantiations as follows. Each non-lexical object-set name s in a label path corresponds to a non-lexical object o instantiated in s. If s is a specialization, o is also instantiated in all of s's ancestors up to the root of the concept hierarchy. For each lexical object set s that appears at the end of a path, the labeled string, itself, is inserted as a member of the object set s. Non-lexical object set names of fields within the same record correspond to the same object for the entire record. For relationship sets, the path among the objects designated by the label path instantiate relationships that connect the objects.

3.2.3 Actively Learning Novel Structures

Active learning is a technique within the field of machine learning to reduce the cost of obtaining labeled training data. The learning system, itself, actively identifies a smaller set of examples whose labels would provide greater utility than a randomly selected set. We may distinguish among active learning methods by their query policies, many of which are described in [53]. These sampling policies all assume that the learning system already knows all candidate labels; querying is a matter of soliciting a label from among these known labels to apply to a given unlabeled example.

In our application, we introduce a new assumption to drive active learning, namely that not all labels are known to the system at the time of a query and that the most helpful query policy is one that is primarily based on novelty detection in that it identifies new structures for which a label is most likely unknown. We are concerned with reaching the minimum possible number of labeled examples which is exactly one label per field. In Sections 3.2.4 and 3.2.5 we explain how to identify new fields using ListReader's regex and HMM wrappers respectively. The basic idea in both cases is to identify when two records are structurally similar to each other except for the insertion of a new field at some identifiable position in one of the two records.

For example, comparing the first and second records in the second list of Figure 3.1, ListReader recognizes that the year of Elizabeth's marriage to Edward ("1801") is a new field and asks for its label. When asked for a label for some identified text, a user responds by altering the form for the list, adding a new entry blank for the field, and filling in this text box by clicking on the appropriate string within the PDF. For the marriage-year example here, the user could add a single-entry blank called *MarriageYear* to the end of the form in Figure 3.2 and click on "1801" in the list in Figure 3.1 (which in our implemented interface sits side-by-side on the screen with the form). This, in turn, alters the ontology in Figure 3.3, adding a new object set *MarriageYear* and a new functional relationship set from *Person* to *MarriageYear*. It also provides a new path expression *Person.MarriageYear* for labeling "1801". Alternatively, the user could connect the new *MarriageYear* field with the *Spouse* field to create a double-column multiple-entry blank, which would produce a ternary relationship involving the person, the spouse, and the marriage date. The *SpouseName* field in Figure 3.2 would be nested inside of *Spouse*, the first of the two columns. The user would also need to click on "1801" to copy it into the first slot of the *MarriageDate* column.

3.2.4 Adaptive Regex Induction

ListReader induces a regex wrapper in three steps: initialization, A* search, and active learning.

During initialization, ListReader begins learning from nothing more than the text of an OCRed page with the fields of the first record of a list labeled by a user. ListReader initializes a new regex to model the text and labels of the first record using a flat sequence of capture groups. Each capture group corresponds to a field or delimiter. Consider, for example, the following labeling of the first record of the first child list in Figure 3.1:

```
<Child.ChildNumber>1</Child.ChildNumber>.
<Person.Name>Andrew</Person.Name>,
b. <Person.BirthDate.Year>1772</Person.BirthDate.Year>.
```

From this labeling, ListReader generates the initial regular expression (regex) in Figure 3.5, a first level generalization of the field delimiters and content.

		Final Regex	
Label (abbrev.)	Initial Regex	RecordType1	RecordType2
RecordDelimiter	(\n)	(\n)	(\n)
ChildNumber	(\d)	(\d)	(\d)
FieldDelimiter	(\.\s)	(\.\s)	(\.\s)
Name	(\w{6,6})	(\w{5,9})	(\w{5,9})
FieldDelimiter			(\s)
Name			(\w{3,8})
FieldDelimiter	(,\sb\.\s)	(,\sb\.\s)	(,\s[bh]\.\s)
BirthDate.Year	(\d{4,4})	(\d{4,4})	([i0-9]{4,4})
FieldDelimiter			([.,]\sd\.\s)
DeathDate.Year			(\d{4,4})
FieldDelimiter	(\.)	(\.)	(\.)
RecordDelimiter	(\n)	(\n)	(\n)

Figure 3.5: Regex Induction for First Child List in Fig. 3.1.

During A* search [34], ListReader generalizes the initial wrapper to produce a set of regexes, one for each record type. That is, ListReader performs an A* graph search once for each line of text below the first record. (ListReader conducts no search if a known regex already matches a record.) Each search traverses a hypothesis space whose nodes are regexes and whose edges are edit operations transforming one regex into another. ListReader uses the first record of the list as the start state in all searches within the same list. Goal states are defined as any regex that matches the entire text of an unlabeled record—the record on which search is performed. The purpose of the search is to find the goal regex with the shortest edit distance from the initial regex.

To generate one regex from another, ListReader applies one of four operators to one capture group position. The operators are insertion, deletion, character class expansion, and word length expansion. The insertion operator inserts a one-word capture group of high generality, e.g., "\S{1,10}", with "Unknown" as its field label. The deletion operator deletes a capture group. The two expansion operators allow a capture group to match a larger class of text than it could before expansion. The character class expansion operator, when applied to a field capture group, moves its text up a shallow hierarchy of character classes, e.g., changing "\w{6,6}" to "\S{6,6}".

The character class expansion operator, when applied to a delimiter capture group, replaces each character in its text with a predefined set of common OCR error substitutions, e.g., it replaces “[.]” with “[.,]”. The word length expansion operator is not used for delimiters, but for fields it expands the upper and lower length bounds of a word by a predetermined increment, e.g., it replaces “\w{6,6}” with “\w{4,9}”.

To control the enormously large search space—millions of states for our small example and hundreds of billions of states for larger ones in our test set—we use an A* search strategy with a carefully designed admissible heuristic. A ListReader A* search iterates over a priority queue of regular expressions. ListReader initializes the queue with a regex based on the initial labeled record (e.g., the Initial Regex in Figure 3.5). On each iteration, the highest-priority regex, r, is dequeued and expanded, meaning that adjacent regexes are generated and added to the queue. The priority[2] of r is $f(r) = g(r) + h(r)$. The $g(r)$ term is the known edit distance from the initial regex to r and the $h(r)$ term is an admissible heuristic estimate of r's remaining distance to a goal. A*'s use of $f(r)$ results in searching paths with the lowest estimated total length. To be admissible, $h(r)$ must never over-estimate the true distance to the nearest goal, ensuring a valuable property of A* search as a whole: it is guaranteed to find the closest goal state first. This makes the search stopping criteria straightforward and the search optimal given the heuristic function. Edit distance is the sum of the edit costs of each operator used to convert one regex into the other. We set the base cost of all operators to 1.0, and we add 0.1 for deletions and insertions that do not occur immediately adjacent to other deletions and insertions, respectively.

Our admissible heuristic $h(r)$ must estimate the remaining distance to the nearest goal from any intermediate regex. Two main ideas guide our heuristic: eliminate redundant search paths and infer how many operator instances are required to convert r into a goal given knowledge of which constituents of r “miss”—do not match with the text being considered. The heuristic we use is the sum of three terms:

[2] Low values have high priority

$$h(r) = order_o(r) + match_{o,t}(r) + missCount_t(r)$$

When an operator is assigned to a specific capture group position within r, we call it an *operator instance* o. The text t is the text of the current record.

The first term, $order_o(r)$, is infinite if o was applied to r “out of order”, and zero otherwise. The same set of operators will ultimately produce the same goal states regardless of their order of application if those operator instances can be applied independently of each other. To generate the RecordType2 regex from the Initial Regex in Figure 3.5, for example, ListReader could have inserted new capture groups after *Name* and after *BirthDate.Year* in either order. Such arbitrariness produces a much larger branching factor for a search space than is necessary for finding the goal state. The $order_o(r)$ term imposes a single left-to-right order of all operator instances that can be applied independently of each other.

The second term, $match_{o,t}(r)$, is zero if o targets a *missing minimal constituent* in r, and infinite otherwise? A *constituent* c is any contiguous subsequence of one or more capture groups within r. A constituent *hits* when it matches at least one substring of t. A constituent *misses* if it fails to hit. A *minimal constituent* is any constituent in r of length l, where l is the size of the smallest constituent in r that misses. The $match_{o,t}(r)$ term restricts applications to constituents that miss, testing constituents against t from small to large until we find a missing minimal constituent. Minimal constituents that already hit need not be targeted as a location of an operator until we attempt to resolve neighboring missing constituents. For example, the *Name* field in the InitialRegex in Figure 3.5 (\w{6,6}) misses when applied to the fourth record in Figure 3.1 because its length must be generalized. Also, the constituent composed of that *Name* field plus the following field delimiter also misses, because a second *Name* field must be inserted between them. However, ListReader cannot recognize the need of the second edit (without extra trial and error) until each of its component capture groups has become generalized enough to hit. The $match_{o,t}(r)$ term provides the most specific location information possible about where the edits might need to happen next to make progress toward a goal state.

The third term, $missCount_t(r)$, gives an admissible heuristic estimate of the number of edit operations needed to reach a goal state. It counts the number of non-overlapping minimal constituents that miss. This is the tightest admissible heuristic we are aware of for this problem. It is not possible to infer the exact number of edits required considering that, if a constituent of r misses, we can only be sure that at least one of its capture groups must be edited by at least one operation. More than one edit may be necessary at the same location, but the necessity of a second edit is not apparent until after the application of the first fails to produce a hit. Moreover, it is often not apparent where a small constituent should hit without testing a larger constituent with more context. Therefore, we test constituents from smallest to largest.

The last of ListReader's three steps is active learning. ListReader queries the user by highlighting the text matched by an *Unknown*-labeled capture group. The user may then modify the form, which provides a new field label and updates the ontology, and then copy the part of the highlighted text that constitutes the field value into the new form field. With the information returned by the user, ListReader creates a new regex consisting of the old regex with the *Unknown* constituent replaced by a new constituent initialized with the newly labeled text. ListReader queries the user once for each inserted constituent among its set of regexes.

If more than one regex matches a string of text, ListReader favors the one with the highest product of edit similarity and match frequency. Edit similarity is one minus the normalized edit distance from the initial regex at the end of the A* search. Match frequency is the number of records in the text matched by the regex divided by the maximum number of records we expect to see in a list (30). Viewing edit similarity as a prior probability and match frequency as an estimate of likelihood, the product score is an approximation of the posterior probability of the applicability of the regex.

After producing a Regex wrapper (e.g., the *Final Regex* in Figure 3.5 as a disjunction of *RecordType1* and *RecordType2*), ListReader executes each record type regex against the unlabeled text, removing segments of text as they match. ListReader uses the labels applied by the regex wrapper to the field text of each record to instantiate the ontology for the list by creating objects and relationships as described in Section 3.2.2.

3.2.5 Adaptive HMM Induction

An HMM:

$$P(S_{1:T}, Y_{1:T}) = P(S_1)P(Y_1|S_1)\prod_{t=2}^{T} P(S_t|S_{t-1})P(Y_t|S_t)$$

represents the joint probability of a sequence of T hidden states $S_{1:T}$ and T corresponding observable state emissions $Y_{1:T}$. The HMM parameters are probabilities in three groups: the marginal probabilities of the first state in the sequence, $P(S_1)$, the *transition model* containing the conditional probabilities of one state given a previous state, $P(S_t|S_{t-1})$, and the *emission model* containing the conditional probabilities of an emission given a state, $P(Y_t|S_t)$. We can set the HMM parameters using maximum likelihood estimation (MLE) from a hand-labeled sequence of observations. Given such a model, we can use the Viterbi algorithm to compute the most probable sequence of states given a new sequence of observations.

For ListReader, we begin by modeling each word in the text as a member of Y using a categorical distribution and each field label as a member of S. During initialization, ListReader collects word and label co-occurrence statistics from the first hand-labeled record and trains the HMM using MLE except for modifications to make the most of sparse data and appropriately model the structure of list text. Figure 3.6 shows an HMM initialized for the first record of the first list of Figure 3.1. Solid lines represent transition or emission distributions trained with whole single counts from the hand-labeled record, and dotted lines represent transitions trained with fractional counts (i.e. non-zero Dirichlet priors). We omit emission model parameters with non-zero priors in this figure for simplicity.

To make the most of our very sparse training data, we employ three techniques. First, we apply a non-zero Dirichlet prior to all parameters that should be non-zero using knowledge of how list text behaves. Dirichlet priors are a versatile form of parameter smoothing that amounts to adding

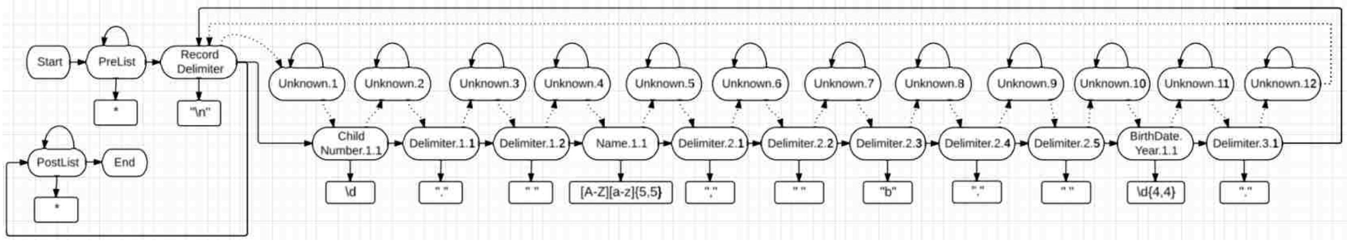


Figure 3.6: HMM Initialized for the First Record in Figure 3.1.

fractional pseudo-counts to the training data where prior knowledge dictates. For example, even though we do not see a 10-character-long *Name* in the first record of Figure 3.1, we know that such a name could exist if we had more training data. We do not want the probability of such a name to be zero. ListReader uses a prior of one divided by the number of words in the list for all words emitted from all states except the *RecordDelimiter* state which can only emit a record delimiter character for the lists we consider.

Second, we use parameter tying in the emission model to force certain states to pool statistics and therefore share a categorical distribution. The states whose parameters we tie are, conveniently, those fields whose labels share the same lexical object set name. Since field labels contain a path in the ontology graph including a leaf node that represents which lexical object set the field text is assigned to, we can use that information as a semantic basis for parameter tying. For example, the emission models for *BirthDate.Year* and *DeathDate.Year* share emission parameters.

Third, we cluster words before performing MLE or applying a trained model to text. To cluster a word, we replace each of its characters with a unique symbol representing which of five character classes it belongs to: uppercase letters, lowercase letters, digits, punctuation, and whitespace. These are represented as regular expressions in Figure 3.6. This word conflation provides a good balance between discrimination and robustness. Word conflation and smoothing also help allow for OCR and other errors which often produce different characters of the same class. For example, "," often changes to "." and vice versa. To avoid drift, we do not further conflate words or adjust word lengths within the emission model beyond the conflated hand-labeled text described here as induction proceeds.

To appropriately model the structure of list text, we modify the HMM in three additional ways. First, we constrain the transition model to have a simple cyclical structure. That is, we avoid transitions from a state to itself and other arbitrary transitions by creating an expanded set of states. This set not only contains separate states for non-list text, record boundaries, field delimiters, and each of the user-specified field labels, but also separate states for individual positions within the word-sequence of each category (hence the numeric state label suffixes in Figure 3.6). These distinct states allow us to give our HMM a strict linear structure from one record delimiter to the next and prevent erroneous loops and short-cuts through the state graph. To produce a precise cyclical structure, we allow the record delimiter state to transition *from* only the pre-list text state and the last state of a record and to transition *to* only the first state of the record and the post-list text state.

Second, we loosen the rigid transition model just enough to allow for deletions and insertions. To allow for deletions, we employ parameter smoothing in the transition model, but only for pairs of states obeying the "total order" specified by the training data. For example, we give a non-zero prior to the *Name-BirthDate.Year* transition and a zero prior to the *BirthDate.Year-Name* transition. To allow for insertions, we add unique *Unknown* states between each pair of field or delimiter states in the HMM, giving higher priors to *Unknown* states that sit at key positions, such as the beginning and ending of delimiters, where knowledge of list structure suggests insertions are more likely to occur. For example, consider applying the HMM in Figure 3.6 to the fourth record of the first list in Figure 3.1. The text "4. " is exactly what the state sequence *ChildNumber.1.1*, *Delimiter.1.1*, and *Delimiter.1.2* expects. Also, ", b. " is exactly what the state sequence *Delimiter.2.1* to *Delimiter.2.5* expects. However, the *Name* state does not expect any of the three intervening tokens "William", " ", or "Lee". Neither of the names are of the expected length and the space is not of the expected character class. However, because of smoothing in the emission models of the intervening states, the *Name* state will match "William" and *Unknown.5* will match " Lee". This is the most probable sequence of matching states because *Unknown.5* appears at the beginning of a long sequence of delimiter states and is therefore given a higher prior. Therefore, ListReader labels " Lee" as *Unknown*.

Third, we model the distinct variability behavior of delimiter and field text in the emission model. Since delimiter text varies less than field text, we prevent ListReader from applying the word conflation described above to any word types found in a known field delimiter. For example, given the delimiter before the birth date in the first record of Figure 3.1, we prevent the word types ",", " ", "b", and "." from being conflated wherever they appear in the page, leaving them as they are.

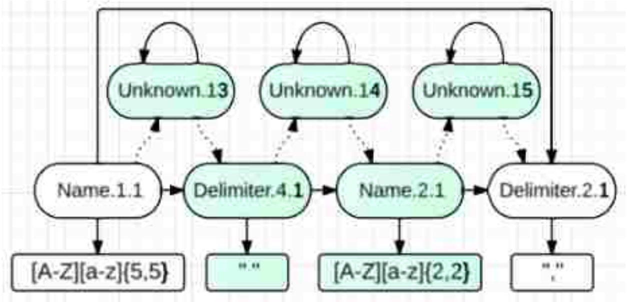


Figure 3.7: New HMM Components (Shaded) after Active Learning for Second Name Field.

During active learning, ListReader queries the user for the labels of each *Unknown*-labeled piece of text. For each query, ListReader trains a new HMM fragment on the newly-labeled text using MLE and the six modifications just described. It inserts this HMM fragment into the original HMM at the same position as the matching *Unknown* state. The probability of the transition entering a new HMM fragment is split with the original transition bypassing the new fragment. It then executes the whole HMM to identify other *Unknown*-labeled text covered by the new HMM fragment before querying the user again. Figure 3.7 shows a fragment produced after querying the user about "Lee". After performing active learning on the whole first list of Figure 3.1, the HMM in Figure 3.6 is almost twice as long, containing new states for *Name*, *DeathDate.Year*, delimiter text before each, and nine new *Unknown* states. When no *Unknown* text remains, ListReader translates labeled fields to instantiated predicates and inserts them into the ontology as explained in Section 3.2.2.

3.3 Experimental Evaluation

A main objective of developing ListReader is to find a way to simultaneously reduce the cost and increase the accuracy of inducing wrappers for lists by taking advantage of list structure. In this light, we evaluate ListReader using the typical accuracy metrics of precision, recall, and F-measure (in particular the F_1-measure), but also using measurements of human labeling cost. For this second objective, we introduce three metrics: *Label Savings*, *Label Efficiency*, and *Active Learning Query F-measure*. We give precise formulas in Table 3.1. Precision is the proportion of field labels produced by the system that are correct. Recall is the proportion of correct field labels that the system produces from those that are possible. F-measure is the harmonic mean of precision and recall. Label savings is the proportion of field labels left unlabeled by the user. For example, the first list in Figure 3.1 contains 25 fields. If the user labels only the three fields in the first record (*Child.ChildNumber* = "1", *Person.Name* = "Andrew", *Person.BirthDate.Year* = "1772"), the savings would be $22/(22+3) = 88\%$. Label efficiency is the harmonic mean of precision, recall, and savings. For both F-measure and label efficiency, we use weights of 1.0, although the weights can be different depending on application needs. Finally, to compute query F-measure, we use the F-measure formula in Table 3.1 based on counts of the following ListReader behaviors: *tp* = the number of times ListReader queries the user when it should (once per field type per list), *fp* = the number of times ListReader queries when it should not, and *fn* = the number of times ListReader fails to query when it should.

Table 3.1: Metrics.

Precision = $p = \frac{tp}{tp+fp}$	Savings = $s = \frac{u}{u+l}$
Recall = $r = \frac{tp}{tp+fn}$	Efficiency = $\frac{w_p+w_r+w_s}{\frac{w_p}{p}+\frac{w_r}{r}+\frac{w_s}{s}}$
F-measure = $\frac{w_p+w_r}{\frac{w_p}{p}+\frac{w_r}{r}}$	

tp = true positives, *fp* = false positives, *fn* = false negatives
u = Number of fields left unlabeled by the user
l = Number of user-labeled fields
w_x = Weight for component x in weighted harmonic mean

A key part of ListReader is a machine-learned sequential labeler. The wrapper formalism listed in Section 3.1 that should be most capable of modeling text in OCRed lists is the Conditional Random Field (CRF), which is a general approach to sequence labeling, achieving state-of-the-art performance in a number of applications [19, 51, 52]. Therefore, we have chosen to compare ListReader to a CRF [46]. To make our labeling task learnable by the CRF, to reduce over-fitting the training data, and generally to ensure a fair test, we tuned its hyper-parameters and selected an appropriate set of word features on a subset of our data. The features we used were (1) the word text itself, and flags indicating which of the following dictionaries the word appears in: (2) given names, (3) surnames, (4) common words with functional part-of-speech including articles, prepositions, and auxiliary verbs, (5) numerals, and (6) initials (a capital letter followed by a period). We also applied the features of immediate neighbors to each token to provide contextual clues. Our dictionaries are large and have good coverage and constitute a greater amount of knowledge engineering than we allow for ListReader.

As test data, we randomly selected sufficient pages from *The Ely Ancestry* [6] to obtain and isolate the text of 60 child lists. These lists contain 3088 non-space word tokens with 1254 field strings to be identified and extracted and a 2% OCR word error rate. Each list contains between 1 and 12 records for a total of 271 records. Records contain a minimum of 2 fields (e.g., “1. Elijah.”) and a maximum of 12 fields (e.g., “5. Elizabeth, b. Dec. 20, 1745; d. June 12, 1777; m. ist, Elihu Halsey; m. 2nd, Frederick Jones.”) and an average of 4.6 fields per record. Records within the same list often differ as much as records from different lists.

To test ListReader accuracy and cost, we hand-labeled the first record of each list and ran ListReader on the list in three variations—A* Regex ListReader as described in Section 3.2.4, HMM ListReader as described in Section 3.2.5, and a simple Regex-HMM ensemble in which, for each record in a list, we take the A* Regex result if one exists, and the HMM result otherwise. We also ran the CRF separately on each list with six variations of labeled training data: the first n and the “best” n records in the list ($1 \leq n \leq 3$), where “three best” is a combination of a longest (1^{st} Best), a least typical (2^{nd} Best), and a most typical (3^{rd} Best) record. From these six, we report here only the two most competitive: the CRF with the best F-measure and the CRF with the best label efficiency.

Table 3.2 gives the results for the measures over all 1254 field values to be extracted from the lists, including both those labeled by the user and those labeled by ListReader. The results in Table 3.2 tell us what accuracy we can expect for the extraction task as a whole and at what cost. Use of the ensemble, which achieves the best accuracy, is motivated by the observation that A* Regex's precision is high when it reaches a goal state but that its recall is low because it sometimes fails to reach a goal state and thus returns no labels for any field in the record.

Table 3.2: Ontology Population Cost Effectiveness (%).

	Prec.	Rec.	F_1	Sav.	Eff.
ListReader A* Regex	**96**	85	**90**	71	**83**
ListReader HMM	**93**	**92**	**93**	70	**84**
ListReader Ensemble	**94**	**94**	**94**	66	**82**
CRF Best F_1	**93**	**92**	**92**	43	67
CRF Best Efficiency	83	72	77	**79**	78

Bold: column max or not sig. lower than the max, $p < .01$, paired t-test

Table 3.3 gives the results when the measures are over just the field values to be labeled by the systems, excluding all hand labeling. These results tell how well the systems generalize from the labeled examples. Over the six CRF variations, not until trained with the "best three" records does the CRF's F-measure approach ListReader's. But in this case the labeling efficiency is much less; plus someone must take the time to select the "three best".

Table 3.3: Wrapper Induction Learning Accuracy (%).

	Prec.	Rec.	F_1
ListReader A* Regex	**95**	80	**87**
ListReader HMM	**89**	**88**	**89**
ListReader Ensemble	**92**	**93**	**92**
CRF Best F_1	83	79	81
CRF Best Efficiency	77	63	69

Bold: column max or not sig. lower, $p < .05$, unpaired t-test

Table 3.4 measures how well A* Regex ListReader and HMM ListReader do at detecting new fields which is our basis for active learning. The 60 lists included 83 opportunities for active learning queries. Table 3.4 shows how many times each ListReader variation asked when it should have (*tp*), asked when it should not have (*fp*), and failed to ask when it should have (*fn*), as well as the query precision, recall, and F-measure.

Table 3.4: Active Learning Query Accuracy (#, %).

	tp	*fp*	*fn*	Prec.	Rec.	F_1
ListReader A* Regex	66	34	17	66	80	72
ListReader HMM	58	45	25	56	70	62

3.4 Conclusions and Future Work

These results suggest that ListReader is a viable way to populate rich ontological structures with data from lists in OCRed historical documents. We can confidently anticipate that for Ely child lists we can extract the information of interest with an F-measure over 90% and with ListReader doing around 70% of the labeling work. The ListReader approach to ontology population converts the extraction and mapping problem into a sequential labeling problem which it then solves by wrapper induction. Both the A* Regex and HMM versions of ListReader perform efficiently and outperform a state-of-the-art sequential labeler (the CRF) in terms of efficiency ($p < .01$). The relative weakness of the CRF is its requiring more training data than ListReader requires to reach the same levels of accuracy. ListReader better leverages the characteristics of list structure with a more tailored machine learning approach.

Although ListReader performs well and outperforms the CRF, we still see room for improvement and thus for much interesting future work. Accuracy results are around 90%, which is comparatively quite good, but the active learning component of ListReader can likely be better—perhaps by leveraging the precision of regular expressions in a synergistic combination with the more flexible HMMs. These improvements should also be valuable as we consider more complex lists: (1) lists split by intervening text or page breaks (e.g., the lists in *The Ely Ancestry* that split across page boundaries), (2) lists nested within other lists (e.g., the child lists nested within the larger family list in Figure 3.1), (3) lists with fields factored out of each record, (e.g., the surname of the children in a family factored out of the child lists in Figure 3.1), and (4) lists whose records describe entities from distinct categories (e.g., business and person addresses intermixed in a city directory).

Besides making ListReader more accurate and able to process more complex lists, we plan to further reduce human effort—not only limiting user involvement to labeling each distinct field of a list only once, but for an entire collection of lists, like all of the child lists in *The Ely Ancestry*. In a bootstrapping effort, we plan to investigate a form of self-supervised wrapper induction to reduce the cost of providing training data for a collection of related lists. ListReader should be able to recognize when it has seen a list similar to a combination of one or more lists or list fragments it has already processed and build both an ontology for the new list and induce a wrapper without human intervention. Ideally, we should benefit from accumulated knowledge resources by no longer needing to create a form to generate the ontology nor to fill in the form to label any of the fields of any of the records.

Chapter 4

Scalable Recognition, Extraction, and Structuring of Data from Lists in OCRed Text using Unsupervised Active Wrapper Induction

Abstract

A process for accurately and automatically extracting asserted facts from lists in OCRed documents and inserting them into an ontology would contribute to making a variety of historical documents Programmatically searchable, queryable, and linkable. To work well, such a process should be adaptable to variations in document and list format, tolerant of OCR errors, and careful in its selection of human guidance. We introduce an unsupervised active wrapper induction solution for finding and extracting information from lists in OCRed text. ListReader discovers lists in the text of an OCRed document and induces a grammar for the internal structure of list records with little document-specific feature engineering or supervision. ListReader then applies the knowledge in this grammar to actively request a limited and targeted set of labels from a user to complete its list wrapper. Lastly, ListReader applies the completed wrapper, encoded as a regular expression, to extract information with high precision from the entire document and automatically maps the labeled text it produces to a rich variety of ontologically structured predicates. We evaluate our implementation on a family history book in terms of F-measure and annotation cost, showing with statistical significance that ListReader learns to extract high-quality data with less cost and lower time and space complexity than a state-of-the-art statistical sequence labeler.

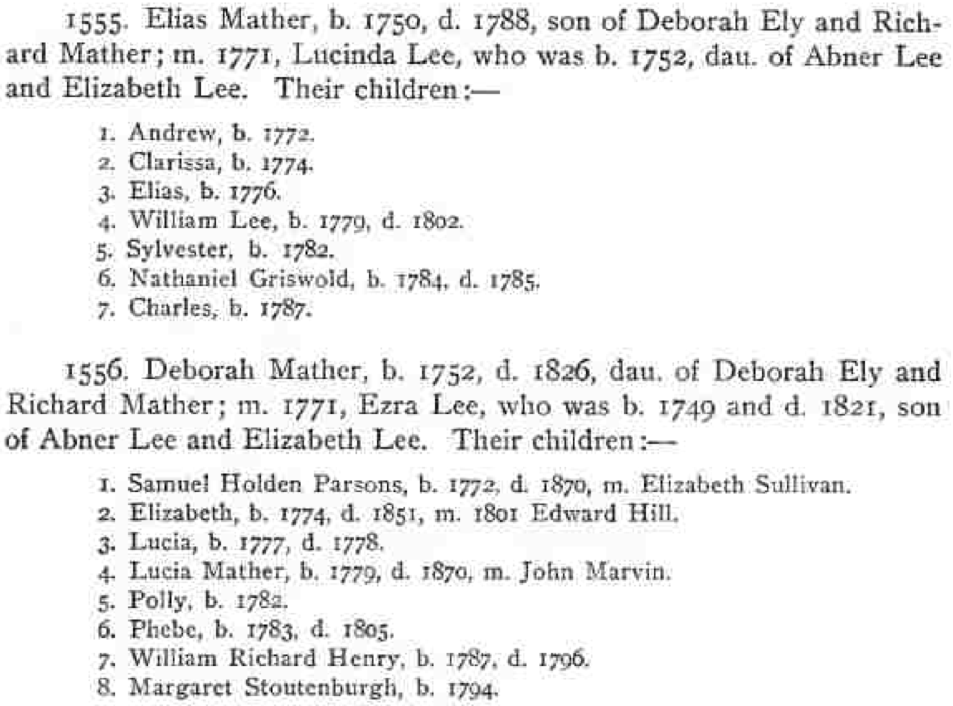
4.1 Introduction

The ability to cheaply and accurately extract information from semi-structured OCRed documents could help a number of seemingly unrelated types of organizations and processes including the following: the electronic filing of paper legal documents, the retrospective conversion of paper books and card catalogs into digital bibliographic databases, the organizing of paper sales receipts in commerce and personal finance smart-phone applications, and the automatic extraction of genealogical data from historical documents in family history research projects. Also, the ability to perform learning and extraction with low time and space complexity is essential when scalability is important—and it is becoming more and more important as “Big Data” projects motivate many to adopt a more scalable approach to their text and data analysis.

To be most useful to downstream search, query, and data-linking applications, the knowledge extracted from text should be expressive, detailed, and structured according to well-established formal conventions. An ontology is an explicit specification of a conceptualization [30]. It is expressive enough to provide a framework for storing more of the kinds of assertions found in lists than the typical output of named entity recognition and most other information extraction work. If we could populate user-specified ontologies with predicates representing the facts asserted in OCRed text, this more expressive and versatile information can better contribute to a number of applications in historical research, database querying, record linkage, automatic construction of family trees, and question answering.

One of the most important text formats is the list. Lists, loosely defined, include any semi-regular repeating pattern of records. Records can be long or short; they can lie in a contiguous block of text or be distributed throughout a document or even a collection of separate but related documents. Little work has been done toward establishing a general approach to extracting information from lists despite how commonly this type of structure appears in text. Some types of books consist almost entirely of lists, such as family history books, city directories, and school yearbooks. The 100,000+ family history books scanned, OCRed, and placed on-line by FamilySearch.org are full of lists containing hundreds of millions of fact assertions about people, places, and events. Figure 4.1

shows a small part of one family history book, a piece of page 154 of *The Ely Ancestry* [6]. It shows two different types of records, members of two lists: parent records and child records.



1555. Elias Mather, b. 1750, d. 1788, son of Deborah Ely and Richard Mather; m. 1771, Lucinda Lee, who was b. 1752, dau. of Abner Lee and Elizabeth Lee. Their children:—

1. Andrew, b. 1772.
2. Clarissa, b. 1774.
3. Elias, b. 1776.
4. William Lee, b. 1779, d. 1802.
5. Sylvester, b. 1782.
6. Nathaniel Griswold, b. 1784, d. 1785.
7. Charles, b. 1787.

1556. Deborah Mather, b. 1752, d. 1826, dau. of Deborah Ely and Richard Mather; m. 1771, Ezra Lee, who was b. 1749 and d. 1821, son of Abner Lee and Elizabeth Lee. Their children:—

1. Samuel Holden Parsons, b. 1772, d. 1870, m. Elizabeth Sullivan.
2. Elizabeth, b. 1774, d. 1851, m. 1801 Edward Hill.
3. Lucia, b. 1777, d. 1778.
4. Lucia Mather, b. 1779, d. 1870, m. John Marvin.
5. Polly, b. 1782.
6. Phebe, b. 1783, d. 1805.
7. William Richard Henry, b. 1787, d. 1796.
8. Margaret Stoutenburgh, b. 1794.

Figure 4.1: Example Text from *The Ely Ancestry*, Page 154.

In this research, we focus on lists in family history books because they contain more rich information and more structural complexity than most other kinds of lists. Records in these lists make many assertions about family relationships and life events that are valuable to family history research but which are not consistent enough in their format for a simple hand-coded script or regular expression to extract. Though each record arranges its information as a sequence of fields and delimiters, the details of this arrangement may differ from one record to another, even within the same list or record type. For example, the fourth child record in Figure 4.1 contains a death date while the preceding child records do not. Considering this type of variation alone—the optionality of fields—the number of record variations we must account for is exponential in the number of fields. In this example, that means the number of variations of the parent records in Figure 4.1, which contain about 18 fields, is at least 262,144 (the cardinality of the power-set of 18 fields is 2^{18}). Furthermore, the different field contents and the OCR errors of otherwise-invariant field delimiters increases the base of that exponential formula, making the total number of possible variations over 387,420,489 even if we assume only two possible variations per field in addition to its optionality.

In addition to record, field, and delimiter variations, we must also account for lack of document structure and metadata. Unlike HTML and other modern text formats, formatting cues are rarely preserved in the output of most OCR engines—cues that humans find invaluable in parsing and understanding lists. OCRed text will generally contain no page layout formatting, no tab-stops, and no font styles. Horizontal spaces of all sizes are collapsed into a single space character. Newline characters and all-caps text are practically the only kinds of formatting preserved. Compared to natural language, semi-structured text often has significant style and structural differences between lists even within the same book, and certainly across books, even in the same genre such as family history. See Appendix A for example page images and OCR text. Even on the same page, we see variations in the structure of the records (especially between the long parent records and the short child records in Figure 4.1).

Despite this complexity, we desire to develop an accurate, low-cost process to extract the rich and diverse kinds of fact assertions from lists in OCRed documents—a process that is robust to OCR errors and variations in list structure. Not only should the process be able to identify corresponding fields among a set of related records, but it should also find all the records in a document without human assistance. Detecting lists automatically could eliminate substantial work for a user, especially in large books or corpora containing mixed content (prose and lists).

We know of three main sources of cost for an information extraction system over a lifetime of use: (1) domain-specific knowledge engineering, (2) input text-specific feature engineering, and (3) labeled text as training examples. To minimize the cost of a specialized information extraction application, we hypothesize that the only truly necessary costs associated with a new topic and text are (1) a minimal specification of the kinds of fact assertions to be extracted and (2) a small amount of machine-specified, hand-labeled text. Our approach, called *ListReader*, relies on minimal domain-specific knowledge engineering, no feature engineering, no hand-labeled training data but a small amount of hand-labeled data to complete the semantic mapping, and no human construction—

or even inspection—of extraction rules. ListReader reduces the cost of the knowledge engineering required to specify fact-assertion templates by allowing the user to specify them in an easy-to-use web-form-building user interface. The idea is that a user specifies the information to be extracted by designing a form—just like we do in practice when we want to "extract" (request) desired information from a person filling out a form. ListReader eliminates other sources of domain-specific knowledge engineering such as the construction of domain lexica, dictionaries, gazetteers, name authorities, or part-of-speech patterns that could be used by machine learning feature extractors or within hand-coded rules. ListReader minimizes the costs associated with text variations and the mapping between text and web form by eliminating the process of feature engineering for our unsupervised machine-learning-based approach and of manually specifying or inspecting regular expressions or other rules which is a common cost in rule-writing approaches. We simplify the process providing system-requested semantic-mapping labels by allowing the user to fill in the user-built web form by simply clicking on the field strings that ListReader automatically finds and highlights. The most apparent and measurable source of cost remaining in our design, and the one that affects scalability the most, is the amount of hand-labeled text for these semantic mappings. Our evaluation of ListReader focuses on reducing the amount of hand-labeled text without sacrificing extraction accuracy.

The primary contribution in this research is a low-cost, scalable, unsupervised active wrapper-induction solution that will discover much of the information recorded in even noisy lists and extract it with high precision as richly-structured data. The wrapper induction algorithm is linear in time and space. The active user interaction is scalable in label-complexity by actively requesting labels that will have the greatest impact on completing the wrapper based on being sensitive to record sub-structure frequencies. Also, the user designs a form at a high-level that in a way an average computer user can do without having to edit regular expressions which requires a more specialized skill set. ListReader relies on no initial labels from the user before becoming effective at querying the user, and it achieves a statistically significant improvement in F-measure as a function of labeling cost compared to two appropriate baselines. The entire grammar induction process is adaptive and robust to record structure variations such as random internal newlines despite needing to be sensitive to the existence of newlines as record delimiters. We believe that this wrapper-induction approach is appropriate for settings in which many input document formats exist, where a separate wrapper should be produced for each format to ensure high-precision, and where the hand-annotation cost budget is low per list or document format.

As a further contribution, we also present a formal correspondence among list wrappers, knowledge schemas, data-entry forms, and in-line annotated text. This correspondence provides the data flow for a process in which a user can easily label plain text for wrapper induction and create a new knowledge schema from the data-entry form itself. It also enables even simple extraction models that produce in-line text labels to extract rich facts from lists and insert them into an expressive knowledge schema. This effectively reduces the knowledge-structure population problem to a sequence labeling problem.

We present our contributions as follows. In Section 4.2, we survey the previous work most closely related to ListReader. In Section 4.3, we give an overview of ListReader wrapper induction and execution from a user's perspective. In Section 4.4, we formalize the correspondence among the four kinds of information: list grammars, knowledge schemas, data-entry forms, and in-line annotated text. In Section 4.5, we introduce a novel linear-time, linear-space unsupervised active wrapper induction algorithm that begins with an unsupervised process of discovering, clustering, and analyzing the internal structure of records, and ends with an interactive labeling process that relies on no initial labels from the user to become effective at querying for additional labels. In Section 4.6, we evaluate the performance of ListReader in terms of precision, recall, F-measure, and field-label cost with respect to a state-of-the-art statistical sequence labeler and a baseline version of ListReader itself. Finally, in Section 4.8, we give current limitations of ListReader, future work, and conclusions.

4.2 Related Work

We identify three categories of work related to ListReader: grammar induction, web-based wrapper induction, and OCRed list reading.

4.2.1 Traditional Grammar Induction

Grammar induction, also known as grammatical inference, in its broadest sense is a large field of research considering the many types of grammars in the Chomsky hierarchy, the probabilistic and non-probabilistic versions of each, the phrase-structure and state machine versions of each, and the number of induction principles, techniques, and input assumptions that are possible (e.g., supervised vs. unsupervised, pre-segmented vs. unstructured input text). Here we focus on the approaches most closely related to ListReader in terms of training criteria and data structures. Most rely on input that is more costly than our approach, including fully supervised labeling of training examples or feature engineering such as part-of-speech tagging.

Wolff [60], [59] applies a combination of the minimum length encoding (MLE) criterion from information theory, multiple string alignment, and search to perform unsupervised grammar induction. Kit [40] also uses an information compression criterion (minimum description length or MDL) to induce a grammar from a Virtual Corpus (VC) compressed into a suffix array. The suffix array improves the time complexity of training from n-gram statistics, but since this data structure relies on a bucket-radix sort, the final time complexity of their grammar induction is $O(n \log n)$. Despite the celebrated properties of MDL as a global optimization criterion, researchers have more recently shown that grammar induction using it as a local optimization criterion are sensitive to the correct calculation of code length [4], [3]. We therefore use a simplified form of MDL and rely on it sparingly in ListReader which requires only $O(n)$ time and space.

Grammars induced for information extraction and wrapper applications are often finite state machines. These state machines often begin as a prefix tree acceptor (PTA) or other ungeneralized structure and are incrementally generalized by merging pairs of states that are selected by a learning criterion (e.g., a Bayesian criterion). This technique has been used to learn both deterministic finite-state automaton (DFA) [25] and hidden Markov model (HMM) grammars [56]. A PTA is a tree-shaped finite state machine built from, and exactly representing, the strings in the input training set. It cannot be used as a starting point when records or strings have not already been segmented. Therefore, this approach will not work for our input text because an OCRed document is not pre-segmented into records. We have found that the suffix tree data structure is a good natural progression from PTAs. Despite the additional work we perform, ListReader's grammar induction has a lower time complexity than the $O(n^4)$ reported by Goan [25].

4.2.2 Web Wrapper Induction

Wrapper induction [41] is the automated process of constructing a model (i.e. a grammar) that can extract and map data from a source document (often tables in HTML web pages) to a uniform, structured data format suitable for querying. The essential difference between the above grammar induction research and wrapper induction is the latter's focus on the final mapping to a queryable database. Another incidental but common difference is that each induced wrapper is specifically designed for one document structure or data source among many, making it potentially more accurate than applying a single, general model to all data sources, but also making it potentially more costly to induce. Our approach has a number of similarities with web wrapper induction research and some key differences.

Tao and Embley [57] describe an efficient means of identifying data fields and delimiters in tables within related "sibling web pages" (pages generated from the same underlying database and HTML template). They look for variability as a sign of data fields and invariability as a sign of delimiters. We apply a similar technique in ListReader to OCRed records, which is in some ways a harder setting because we must also discover and segment the records before aligning them and because the field-and-delimiter sequences will not align as consistently within an OCRed list as they do within a born-digital, machine-generated set of HTML tables. Embley, Jiang, and Ng [23] automatically determine which HTML tags are record separators using a set of heuristics combined using Stanford Certainty Theory. The ListReader approach described below can find record boundaries without supervision but does assume that all record boundaries contain a newline which is always the case in our chosen document genre.

A few wrapper induction projects target semi-structured text (including lists) in HTML documents. Choices in wrapper formalism include sets of left and right field context expressions [41], [5], xpaths [20], finite state automata [43], and conditional random fields [22], [32]. These formalisms generally rely on consistent landmarks that are not available in OCRed lists for three reasons: OCRed list text is less consistently structured than machine-generated HTML pages, OCRed text does not contain HTML tags, and field delimiters and content in OCRed documents often contain OCR and typographical errors. None of these projects address all of the steps necessary to complete the process of the current research such as list finding, record segmentation, and field extraction.

The wrapper induction work most closely related to ListReader is IEPAD [15]. IEPAD consists of a pipeline of four steps: token encoding, PAT tree construction, pattern filtering, and rule composing. Like ListReader, IEPAD must deal with a trade-off between coarsely encoding the text to reduce the noise enough to find patterns and finely encoding the text to maintain all the distinctions specified by the output schema. Also, PAT trees are related to suffix trees and share similar time and space properties. However, we note some important differences. ListReader must use a very different means of encoding (conflating) text than IEPAD so it can preserve more fine grained structure. This is because, given OCRed text, ListReader cannot rely on HTML tags to delimit fields and newlines to delimit records, and nearly any type of string can be a field delimiter. The field content of OCRed lists appears to contain more variability than the tabular data of HTML pages, and yet fewer consistent cues are available in performing alignment. IEPAD apparently cannot extract fields that are not explicitly delimited by some kind of HTML tag. Also, it appears that the IEPAD user must identify pages containing target information. A ListReader user does not need to do so. The IEPAD user is required to select patterns because the system may produce more than one pattern for a given type of record. ListReader automatically selects patterns among a set of alternatives using a simplified MDL criterion. IEPAD users must also provide labels for each pattern, which is similar to the work ListReader users must do, but is likely more difficult than to label the actual text of a record because it forces the user to interpret the induced patterns instead of the original text. ListReader also minimizes the amount of supervision needed to extract a large volume of data by integrating an interactive labeling process into grammar induction, something IEPAD does not do. Lastly, neither IEPAD nor any of the above research has been applied to recognizing or extracting information from lists in OCRed text.

4.2.3 Lists in OCRed Documents

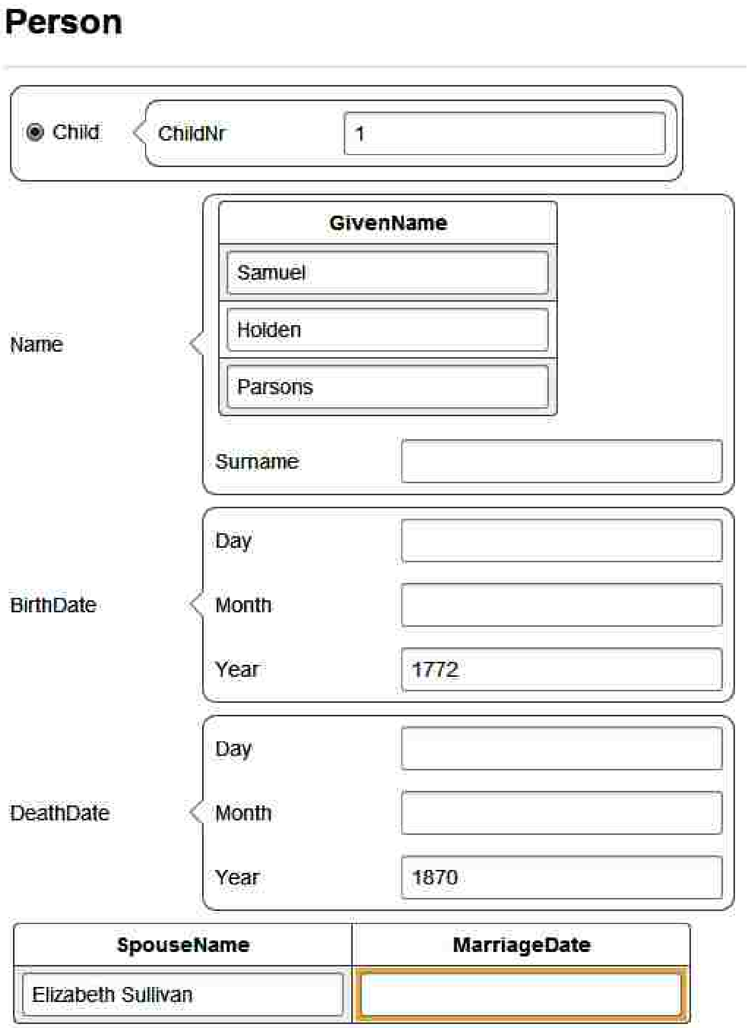
Most systems that extract information from OCRed lists limit their input to specific kinds of lists or records, assume pre-segmented records, or do not apply induction techniques that are adaptable and scalable. Belaïd [7], [8] and Besagni, et al. [9], [10] extract records and fields from lists of citations, but rely heavily on hand-crafted knowledge that is specific to bibliographies. Adelberg [1] and Heidorn and Wei [35] target lists in OCRed documents in a general sense. They, however, use supervised wrapper induction that we believe is less adaptive or scalable than our methods when encountering the "long tail" of list formats. They do not evaluate cost in combination with accuracy as we do and the extracted information is limited in ontological expressiveness (which is true of all existing work in grammar and wrapper induction of which we are aware).

We conclude this section by comparing the ListReader approach described here with our own previous work in this area. In [48] we present both Regex- and HMM-based wrapper induction techniques. While both work well at extracting information from some lists, they have limitations. Both approaches assume that the user will find the lists of interest and label the first record of each list before wrapper induction or active learning begins. Since these approaches assume that a list is a contiguous block of records, each wrapper induction process can induce a wrapper for only one—possibly very small—contiguous block of records at a time, potentially requiring the user to label the same types of fields and records again whenever they appear in another block of records. Also, these approaches rely on the user labeling every field in a record so that ListReader can know which parts of the record (the fields) are variable across records and which parts (the delimiter) should remain more or less constant. Finally, in the case of the regex wrapper induction, its approach to handling the combinatoric problem mentioned above is to explicitly search over this exponentially-sized hypothesis space using an A* search over record variations. Despite a custom admissible search heuristic that we designed for this problem, this regex induction approach is unable to scale up to the search space of the longest records (e.g., the parent records in Figure 4.1). The wrapper induction approach described in the remainder of this paper overcomes all of the above limitations.

4.3 ListReader Overview

ListReader populates a schema structure (an *ontology*) with data it takes from lists in a text document. A user U begins by selecting an OCRed document (e.g., a family history book) and places the pages in a directory. Figure 4.1 shows some of the text of *The Ely Ancestry*. Other pages from this 830-page family history book are in Appendix A. In our implementation the pages are PDF images with an accompanying OCRed layer of text.

Using our form-builder interface, U next creates a form, which specifies the information of interest to be extracted from the text. Figure 4.2 shows an example in which the information found in the child records in *The Ely Ancestry* is specified. Typically, a record names a *Child*, who *is-a Person* and who may have some or all of the following properties: a *ChildNr*, a *Name* consisting of one or more *GivenName*s and possibly a *Surname*, a *BirthDate* and *DeathDate* both consisting of a *Day*, *Month*, and *Year*, and one or more spouses with a *SpouseName* and a *MarriageDate*. U may also specify other forms for gathering information. Although it is common to specify the forms in advance, U can instead build them along the way as information of interest is encountered. In our current implementation, we have both modes of form building, and indeed their combination so that U can specify a form in advance but then add to it along the way. We do, however, only allow field additions—dynamic form reorganization and the disposition of captured information when users delete form fields are beyond the scope of the project.



Person

● Child

ChildNr: 1

Name

GivenName: Samuel
Holden
Parsons

Surname:

BirthDate

Day:

Month:

Year: 1772

DeathDate

Day:

Month:

Year: 1870

SpouseName	MarriageDate
Elizabeth Sullivan	

Figure 4.2: Filled-in Form for Samuel Holden Parsons Record.

The metaphor of form fill-in for obtaining information is familiar to most users, as is form creation from the basic set of primitives we provide. Our form primitives include the following: a single-entry form field to accept single values (e.g., the *ChildNr* “1” in Figure 4.2), a multiple-entry form field to accept multiple entries (e.g., the *GivenName* field in Figure 4.2 with three entries “Samuel”, “Holden”, and “Parsons”), a two-or-more column multiple-entry form field to accept n-ary relationships (e.g., the ternary relationship among a person, spouse, and marriage date in Figure 4.2), and radio buttons and check boxes to respectively accept one role or several role designations (e.g., the radio button to designate the *Child* role or subclass of *Person* in Figure 4.2). The nesting of form fields provide for relationships among the form elements whose leaf elements are for text objects. The title of the form, *Person* in our example, designates the main object—the object the record describes.

Once U loads a document into a directory, ListReader can begin its work. We present the details of the grammar induction pipeline in Section 4.5. Here we sketch the basic idea. The process starts by conflating or abstracting strings in the text. With the text conflated, ListReader finds, clusters, and aligns the most common types of records—records whose sequence of conflated text are identical. Since our goal is to align multiple field strings in text with corresponding ontology concepts, ListReader must also—explicitly or implicitly—align the fields in one record with the equivalent fields in other records. Variations in the text—both intentional and unintentional with regard to the author—make the alignment more difficult. The difficulty can always be resolved with higher cost by asking U to label more examples, but we wish to minimize this. Label efficiency is therefore a matter of making confident alignments among field strings where the alignments are robust to variations in the text. When ListReader is confident that two or more strings have the same label, it can request a label from U for one of the strings in the cluster and then automatically and confidently apply it to the others. In a large book such as *The Ely Ancestry*, ListReader creates hundreds of aligned clusters containing as many as a thousand or more records in the largest aligned clusters.

ListReader performs all of this work automatically—without user involvement—relying on the text itself to provide insight into list structure before starting the supervised part of wrapper induction. Indeed, ListReader, at this point in the process, can produce full extraction rules. For example, for the cluster containing the record "`1. Andrew, b. 1772.`" in Figure 4.1, which also would include the 2nd, 3rd, 5th, and 6th children of Elias Mather and the 5th child of Deborah Mather, and hundreds, if not thousands, of other records in *The Ely Ancestry*, ListReader would produce a regular-expression rule like:

```
([\n])([\d]{1})(\.)([ \n])(([A-Z]+[a-z]+|[A-Z]+[a-z]+[A-Z]+[a-z]+))
(,)([ \n])(b)(\.)([ \n])([\d]{4})(\.)([\n])
```

ListReader does not know, however, which capture groups contain the information to be extracted or to which form fields the captured text applies. To make this determination, ListReader begins the interactive part of wrapper induction to obtain labels for capture groups. It selects one of the strings in a cluster and displays the page containing the selected record and identifies the part of the text U should label. ListReader also displays an empty form like the one in Figure 4.2 along side the page. If U is working with only one form, ListReader displays it; otherwise, U must select, augment, or build a form for the data. Supposing, for example, that ListReader asks U to label the first record in the second child list in Figure 4.1 with the form in Figure 4.2, then in our implementation, ListReader would display the empty form beside the page, and U would fill it in by clicking on the words in the text for each field in the form, yielding the filled-in form in Figure 4.2.[1] Given the filled in form, ListReader knows how to label the capture groups in the regular expression for the cluster in which the Samuel Holden Parsons record appears.

From an empty form, ListReader creates the schema of an ontology which we represent as a conceptual-model diagram. From the form in Figure 4.2, for example, ListReader creates the diagram in Figure 4.3. From the form primitives, ListReader can construct and fill in ontology schemas with the following five points of expressiveness: (1) textual vs. abstract entities (e.g., *GivenName*("Samuel") vs. *Person*($Person_1$)); (2) 1-many relationships in addition to many-1 relationships so that a single object can relate to many associated entities instead of just one (e.g., a *Name* object in Figure 4.3 can relate to several *GivenName*s but only one *Surname*—the arrowhead in the diagram on *Surname* designating functional, only one, and the absence of an arrowhead on *GivenName* designating non-functional, allowing many); (3) n-ary relationships among two or more entities instead of strictly binary relationships (e.g., in Figure 4.3 we can have *Person-SpouseName-MarriageDate*($Person_2$, "Edward Hill", "1801") for the second child, Elizabeth, in Deborah Mather's family in Figure 4.1); (4) ontology graphs with arbitrary path lengths from the root instead of strictly unit-length as in named entity recognition or data slot filling (e.g., *Person.BirthDate.Year* in Figure 4.3); and (5) concept categorization hierarchies, including,

[1] Note that the highlighted *MarriageDate* field in Figure 4.2 is the field of focus awaiting a click on a marriage date, but none is given, so U leaves the field blank.

in particular, role designations (e.g., *Child is-a Person*). This expressiveness provides for the rich kinds of fact assertions we wish to extract in our application.

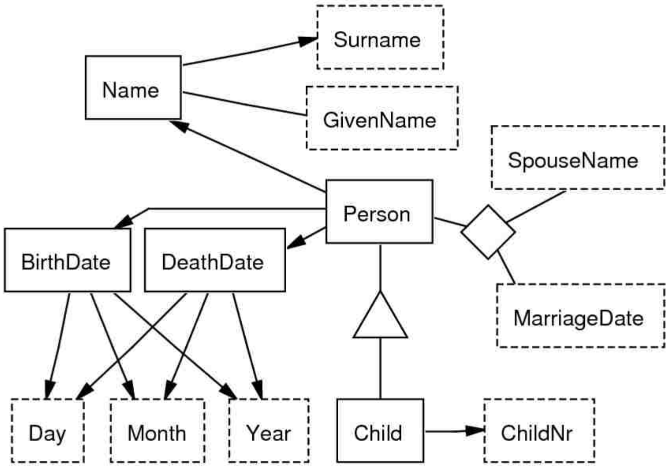


Figure 4.3: Ontology Specifying Information of Interest.

ListReader can also use the information obtained from the filled-in form to label the fields within the text as Figure 4.4 shows. Label names correspond to fields in the form. ListReader assigns these labels to corresponding capture groups of the regex wrapper so that it can label additional text throughout the input document. These labels also guide ListReader in mapping the labeled field strings to predicates. Table 4.1 shows how ListReader populates the ontology schema in Figure 4.3 for the labeled text in Figure 4.4—thirteen unary predicates, nine binary predicates, and one ternary predicate. Since the marriage date is missing, the ternary predicate includes a marked null ($\perp_1$) as a placeholder.

```
<Child.ChildNr>1</Child.ChildNr>
. <Person.Name.GivenName>Samuel</Person.Name.GivenName>
<Person.Name.GivenName[2]>Holden</Person.Name.GivenName[2]>
<Person.Name.GivenName[3]>Parsons</Person.Name.GivenName[3]>
, b. <Person.BirthDate.Year>1772</Person.BirthDate.Year>
, d. <Person.DeathDate.Year>1870</Person.DeathDate.Year>
, m. <Person.(MarriageDate,SpouseName)>Elizabeth
Sullivan</Person.(MarriageDate,SpouseName)>.
```

Figure 4.4: Labeled Samuel Holden Parsons Record.

Table 4.1: Predicates Extracted from the Samuel Holden Parsons Record.

Person($Person_1$)	*MarriageDate*($\perp_1$)
Child($Person_1$)	*Child-ChildNr*($Person_1$, “1”)
ChildNr(“1”)	*Person-Name*($Person_1$, $Name_1$)
Name($Name_1$)	*Name-GivenName*($Name_1$, “Samuel”)
GivenName(“Samuel”)	*Name-GivenName*($Name_1$, “Holden”)
GivenName(“Holden”)	*Name-GivenName*($Name_1$, “Parsons”)
GivenName(“Parsons”)	*Person-BirthDate*($Person_1$, $BirthDate_1$)
BirthDate($BirthDate_1$)	*BirthDate-Year*($BirthDate_1$, “1772”)
Year(“1772”)	*Person-DeathDate*($Person_1$, $DeathDate_1$)
DeathDate($DeathDate_1$)	*DeathDate-Year*($DeathDate_1$, “1780”)
Year(“1780”)	*Person-SpouseName-MarriageDate*($Person_1$, “Elizabeth Sullivan”, $\perp_1$)
SpouseName(“Elizabeth Sullivan”)	

4.4 Representation Correspondences

To automate much of ListReader processing, we establish mappings among three types of knowledge representation: (1) HTML forms (e.g., Figure 4.2), (2) ontology structure (e.g., Figure 4.3), and (3) in-line labeled text (e.g., Figure 4.4). This effectively reduces the ontology population problem to a sequence labeling problem. We formalize the correspondence among the three representations with several definitions, which immediately yields the mappings among them.

List A *list* L is an ordered set of strings, not necessarily contiguous within an input document, that share a record template—the same sequence of fields and delimiters.

The list:

```
1. Andrew, b. 1772.
2. Clarissa, b. 1774.
3. Elias, b. 1776.
5. Sylvester, b. 1782.
7. Charles, b. 1787.
5. PoUy, b. 1782.
```

taken from Figure 4.1 is an example. The record template consists of three fields—a child number, name, and birth year—and four delimiters—“<*newline*>”, “`. `”, “`, b. `”, and “.<*newline*>”.

For a list L, we seek to establish both an ontology O for the fields of L and their interrelationships and to populate O with the fact assertions stated in L.

Fact A *fact* is an instantiated, first-order, n-ary ($n \geq 1$) predicate, asserted to be true.

Ontology An *ontology* is a triple (O, R, C): O is a set of object sets; each is a one-place predicate; each predicate has a *lexical* or a *non-lexical* designation (instantiated, respectively, only by value constants and only by object identifiers). R is a set of n-ary relationship sets ($n \geq 2$); each is an n-place predicate. C is a set of constraints: referential integrity, cardinality, and generalization/specialization.

An ontology O can be rendered as a hypergraph (e.g., Figure 4.3). Lexical object sets appear as boxes with dashed lines, and non-lexical object sets appear as boxes with solid lines. The nodes of an ontology hypergraph are of two types: (1) an object set not in a generalization/specialization hierarchy (e.g., all object sets except *Person* and *Child* in Figure 4.3), and (2) a generalization/specialization hierarchy in its entirety, denoted by the object-set name of any one of the object sets in the hierarchy (e.g., the *Child is-a Person* generalization/specialization hierarchy in Figure 4.3). The edges of O are sets of two or more nodes[2] and represent sets of relationships among concepts. The lines connecting object sets in Figure 4.3 are binary relationship sets. For an n-ary relationship set ($n > 2$), we add a diamond at the connecting point of the three or more connecting lines, which distinguishes it visually from crossing lines (e.g., the ternary relationship set among *Person*, *SpouseName*, and *MarriageDate in Figure 4.3*). Referential integrity must always hold so that in a populated ontology, objects related in a relationship exist in their respective object sets. In Table 4.1, each object in a relationship-set predicate is also in its object-set predicate. Cardinality constraints allow for restrictions on relationship sets (e.g., *functional* constraints, designated by an arrowhead on the range side, restrict the relationship set to be a (partial) function from domain

[2] Because edges may relate more than two nodes, ontology diagrams are *hypergraphs* rather than graphs for which all edges connect exactly two nodes.

object set to range object set). In Figure 4.3 functional constraints restrict, for example, a *Child* to have at most one *ChildNr* and a *Person* to have at most one *BirthDate*. Generalization/specialization hierarchies constrain specialization object sets to be subsets of their generalization object sets.

Path A *path* in an ontology is a sequence of nodes such that consecutive nodes reside in the same edge.

In the path <*Person, DeathDate, Month*> in Figure 4.3, for example, {*Person, DeathDate*} is an edge as is {*DeathDate, Month*}. For path <*Name, Person, MarriageDate*>, the edges are {*Person, Name*} and {*Person, SpouseName, MarriageDate*}, and for path <*Name, Child, ChildNr*}> (= <*Name, Person, ChildNr*>) the edges are {*Person, Name*} and {*Child, ChildNr*} where *Person* and *Child* name the same node—the generalization/specialization hierarchy *Child is-a Person*.

List Ontology A *list ontology* for a list L is an ontology that (1) has a non-lexical object set, called the primary object set or root object set, whose object identifiers denote the objects represented by the records in L, one for each record and (2) has the following restrictions: (a) distinct object sets have distinct names, (b) relationship sets may be constrained to be functional but otherwise have no cardinality constraints, (c) all generalization/specialization hierarchies have a single root and consist of all non-lexical object sets, and (d) for each lexical object set s, at least one non-cyclic path p connects the root object set r with s such that all object sets in p are non-lexical (except for s, itself).

The ontology in Figure 4.3 is a list ontology. The primary object set is *Person*. A non-cyclic path exists from *Person* to every lexical object set. Some paths are immediate (e.g., <*Person, SpouseName*>, <*Person, MarriageDate*> and <*Person, ChildNr*>), and all the rest have intermediate non-lexical nodes (e.g., <*Person, Name, GivenName*>). *Day*, *Month*, and *Year* all have two paths (e.g., for *Year* the two paths are <*Person, BirthDate, Year*> and <*Person, DeathDate, Year*>.

List Form A *list form* corresponds precisely to a list ontology o: the primary object set is the form title, and each path of o and each specialization within a generalization/specialization hierarchy of

o is represented by a nesting of fields—single-entry form fields for functional parent-child edges, single-column multiple-entry form fields for non-functional binary parent-child edges, $n-1$-column multiple-entry form fields for n-ary parent-child edges; and radio-button or check-box fields for specializations.

Observe that the form in Figure 4.2 corresponds precisely to the list ontology in Figure 4.3. The path <*Person, Name, Surname*>, for example, has the single-entry form field *Surname* nested under the single-entry form field *Name*, which is nested under the form title, *Person*. The paths <*Person, SpouseName*> and <*Person, MarriageDate*>, which are both part of a 3-ary relationship set, are nested as a 2-column multiple-entry form field under Person. *Child* is a specialization nested under *Person*.

Based on the correspondence of a list form and a list ontology, the data instances in the fields of a list form immediately map to object and relationship sets in a list ontology. Table 4.1 gives the mapping for the data instances in the filled-in form in Figure 4.2.

In addition to providing a mapping of data in a form field to an ontology, list forms also provide a way to label instance data in a list record. Given the form in Figure 4.2, we can label the first child record of Deborah Mather in Figure 4.1 by copying the strings in the document into the form. In our ListReader implementation, we copy by clicking on a string when the focus is on the form field into which we wish to copy the string. Thus, ListReader knows exactly where in the document the strings are located and can also generate and place an in-line label in the document itself as Figure 4.4 shows. Observe that the labels in Figure 4.4 are paths from the primary object set to the lexical object set into which the text string is to be mapped. Whenever a multiple-entry form field appears in the path, an instance repetition number is appended to designate to which repetition the instance belongs for all instances beyond the first. Thus, for example, in Figure 4.4, *Person.Name.GivenName* is the label for "Samuel", the first given name, and *Person.Name.GivenName[2]* is the label for "Holden", the second given name.

Field Instance Label Let s be a string in a document to be labeled as belonging to field f in a list form with corresponding list ontology o. Let p be a path from the root object set r of o to the lexical object set corresponding to f. Let p' be p augmented with repetition numbers for multiple-entry form fields on the path p—no augmentation for the first field in a multiple-entry form field, “[2]“ for the second field, “[3]” for the third field, etc. Then, p' is a *field instance label* for a string s.

Observe that a field instance label specifies exactly which form field is to be filled in with the labeled string. Hence, given the mapping of form fields of a list form corresponding to a list ontology, the label specifies a mapping of a labeled string to the ontology. We thus see that ListReader’s task is to find records for a list ontology o and label the strings in the records with respect to o. ListReader does so by inducing a wrapper—in ListReader’s case, a labeler of strings in list records.

4.5 Unsupervised Active Wrapper Induction

Our approach to wrapper induction is a novel combination of the fundamental ideas of both unsupervised learning and active learning. ListReader is *unsupervised* in that it induces a grammar without labeled training data and does not alter this grammar after it makes *active* requests of the user for labels which it receives and assigns to existing elements of the grammar. ListReader follows the principles of the active learning paradigm [38] in that it uses this structural model to request labels for those parts of the known and unlabeled structure that will have the greatest impact on the final wrapper.

ListReader’s unsupervised grammar induction must answer a number of questions from the unlabeled text such as: “Where are the lists and records in the input text?”, “What are the record and field delimiters?”, and “How are records composed of fields and field delimiters?”. The grammar induction answers these questions from unlabeled text in a pipeline of steps, with later questions building upon previous answers. It resorts to labeled text only when necessary—at the end. This is an adaptive strategy because it asks questions of the input text instead of making unjustified assumptions about the text. This is a cost-effective strategy because it first looks for answers in unlabeled text. The questions ListReader asks of the unlabeled text improve its understanding of the structure of the document before the user provides any labels. ListReader then interprets those labels with respect to the structure it has identified. This is a scalable strategy both because its execution time and space bounds are linear and because of the limited number of requests it makes of a user.

ListReader's full unsupervised active grammar induction pipeline includes the following 13 steps. (Steps marked with an asterisk are optional. In a run, ListReader either executes all or none of these optional steps.)

1. Input requirements
2. Conflation parsing
3. Suffix tree construction (1)
4. Record selection
5. Record cluster adjustment
6. * Field group delimiter selection
7. * Field group template construction
8. * Field group parsing
9. * Suffix tree construction (2)
10. * Revised record selection
11. Regex construction
12. Active sampling
13. Wrapper output

In the following subsections we give the details of each step, illustrate with a short example, and analyze time and space complexity.

4.5.1 Input Requirements

ListReader requires three inputs: (1) a book, (2) an array of conflation rules in order of application, and (3) a set of record pattern properties. Although not a required input, ListReader users may predefine one or more forms; alternatively, users may define and augment forms during active sampling. All required inputs except the book have default values which we have set empirically using development data (*The Ely Ancestry*) and held fixed through the blind evaluation with the *Shaver-Dougherty Genealogy* [54]. ListReader currently takes a book in either Adobe PDF format containing a layer for OCR text and a layer for the original scanned image or a sequence of one or more plain text files.[3] ListReader reads in the unlabeled OCR text of the whole book as a single sequence of characters and later displays individual page text and images (if available) for each query of active sampling. Conflation rules define how to abstract text to help align patterns. The record pattern properties come into play in record selection and include the following: a set of possible record delimiter characters, delimiter frequency, minimum pattern count, minimum pattern length, and numeral and capitalized word count. We explain the default set of conflation rules and record pattern properties in more detail, below, in the context of the processes that use them.

This step in the pipeline contributes $O(t)$ in both space and time as it reads in the text of the book. The other inputs contribute only small constants to time and space complexity.

4.5.2 Conflation Parsing

ListReader converts the input text into an abstract representation using a small pipeline of "conflation rules". In addition to tokenizing the input text, these rules have a purpose similar to both the pyramid processing method in computer vision [2] and phonetic algorithms like Soundex in searching for historical variants of names [36]. In these cases, we wish to "blur out" superfluous and problematic distinctions within equivalence classes, such as field content variations and OCR errors. Furthermore, we extend these conflation rules to larger phrasal variations, e.g., person names of varying length. A single application of a single rule replaces a small string with a small parse tree. The sequence resulting from all applications of one rule is the input for the application of the next rule in the conflation pipeline. The final sequence of the roots of these parse trees and any remaining non-

[3] Any document containing text could be used as input.

conflated characters form a new sequence that is easier to cluster and align in downstream steps and is therefore the input of subsequent steps in the main ListReader pipeline.

Each conflation rule must describe (1) the pattern of input text that it matches, including content and pre- and post-context (if needed) and (2) the resulting output symbol to replace the matching content. Currently, we have established the following conflation rules, given in their order of application.

1. *Split Word*: Concatenates two alphabetic word tokens that are separated by a hyphen and a newline. The resulting symbol omits the hyphen and newline, concatenates the two words, and produces the same output symbol that the *Word* rule (below) would have produced on the concatenated text by conflating the upper and lower characters of the word.

2. *Numeral*: Finds any contiguous sequence of one or more digits and replaces each digit with “Dg” and formulates a symbol for the contiguous sequence. For example, the input text “`1776`” becomes the complete symbol “`[DgDgDgDg]`”. (Square brackets mark the beginning and end of a symbol.)

3. *Word*: Replaces any contiguous sequence of alphabetic characters with a symbol that retains the order in which uppercase and lowercase characters appear within the word. We replace all contiguous sequences of uppercase letters in the original text with one symbol (“`Up`”) and all contiguous subsequences of lowercase letters with a different symbol (“`Lo`”). Additionally, we conflate the pattern “`[UpLoUpLo]`” with the pattern “`[UpLo]`”, because the former is almost always either a surname (like “`McLean`”) or a capitalized word containing an internal OCR error, e.g., “`PhUip`” instead of “`Phillip`”.

4. *Space*: Replaces common horizontal space characters (“ ”) and newlines (“`\n`”) with a generic space symbol “`[Sp]`”.

5. *Incorrect Space*: Removes spaces that occur on the “wrong side” of certain punctuation characters because of an OCR error. Incorrect spaces include spaces immediately before

phrase-ending punctuations like period (“` .`”), comma (“` ,`”), semi-colon (“` ;`”), and colon (“` :`”); and spaces just inside grouping symbols (“`( `” or “` )`”).

6. *Word Repetition*: Replaces sequences of capitalized words delimited by “`[Sp]`” with a new symbol “`[UpLo+]`”.

Conflation rules have two functions. First, they determine frequently occurring constituency patterns of characters, words, and phrases. Some of these constituency structures are preserved in the final grammar (regex) as nested capture groups because the lower-level constituents often require distinct labels. For example, the individual words in a Word Repetition may require separate labels such as $GivenName$ and $Surname$. Second, they remove superfluous distinctions in the input text (like OCR errors and field variations) that prevent pattern-matching and alignment such as the space in Incorrect Space or the individual characters in Numeral and Word. ListReader does not need to preserve these lower-level constituents as regex capture groups.

Figures 4.5 and 4.6 respectively contain the parse tree sequences for

`Deborah Ely and Rich-\nard Mather ;`

and

`\n5. PoUy , b. 1782.\n6. Phebe, b. 1783`

from Figure 4.1. Notice that before conflation, the strings “`\n5. PoUy , b. 1782`” and “`\n6. Phebe, b. 1783`” are not equal and do not align, but after conflation they align easily because of their equal sequence of parse-tree roots:

“`[Sp][Dg].[Sp][UpLo+],[Sp][Lo].[Sp][DgDgDgDg]`”.

We can observe the shared pattern in the sequence of root nodes in the parse in Figure 4.6, connected by dashed, right-pointing arrows.

This step adds $O(t)$ space and time because it iterates over the length of the text once for each of a small, fixed number of conflation rules; each conflation rule can add no more than one new symbol per input character.

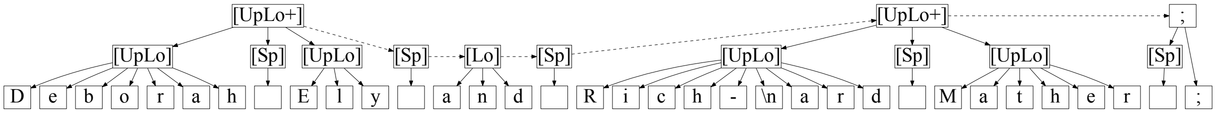


Figure 4.5: Conflation Tree of Text “`Deborah Ely and Rich-\nard Mather ;`”.

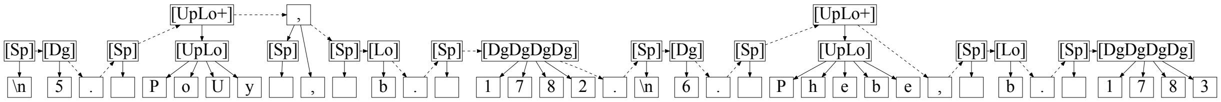


Figure 4.6: Conflation Tree of Text “`\n5. PoUy , b. 1782.\n6. Phebe, b. 1783`”.

4.5.3 Suffix Tree Construction (1)

Once simplified, ListReader can find record-like patterns in the input text by searching for subsequences that repeat. To find these text patterns efficiently, ListReader first constructs a suffix tree from the conflated text sequence.

Suffix Tree Given an input text that is t symbols long (plus a special end symbol not in the input text, e.g., $), a *suffix tree* is a compact data structure representing all $t+1$ suffixes of the text by paths to the tree’s $t+1$ leaf nodes. Each edge in the suffix tree is labeled by the substring of symbols it represents, the number of times that string occurs in the input text, and the beginning offsets of each string occurrence (starting at text offset 0). Each concatenation of the substring labels along a path from root to leaf is one of the $t+1$ suffixes.

To illustrate, Figure 4.7 is an example suffix tree built from a small, non-conflated part of our running example text: “`\nElias.\nElizabeth.`”. In our example, the second branch (second down from the top) from the root (left-most node) represents the two occurrences of the string “Eli” found at offsets 1 and 8 in the input text. Branches descending from a node represent all the different suffixes of those substrings. For example, the branches descending from “Eli” include the rest of the names “Elias” and “Elizabeth” as well as the rest of the input string following each. The number of leaf node descendants of an interior node equals the number of occurrences of the shared prefix represented by that node. Therefore, two leaf nodes descend from the “Eli” edge.

The root of the suffix tree represents strings that share the empty string as a prefix; therefore its descendants include every possible suffix of the input text.

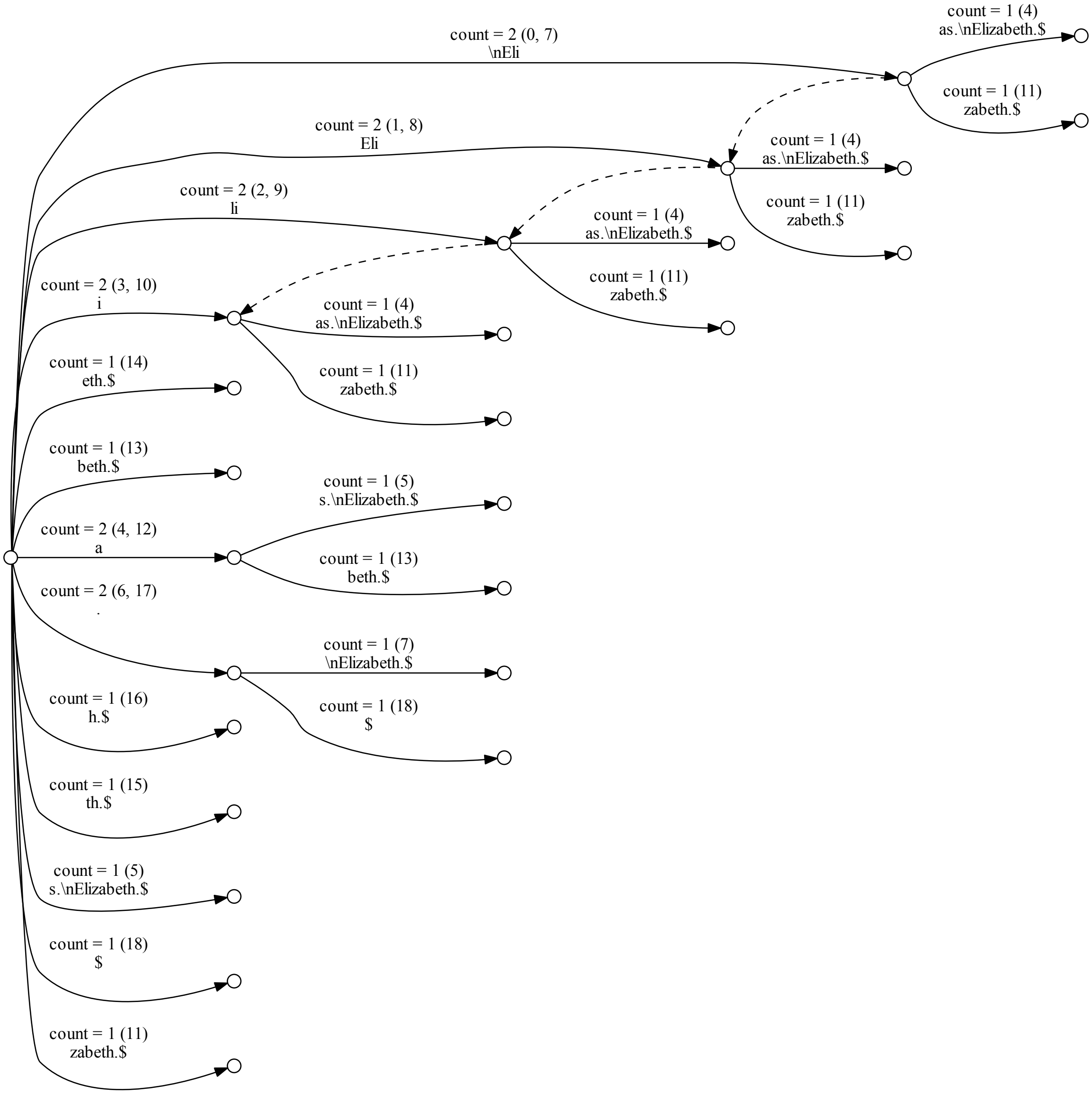


Figure 4.7: Suffix Tree of Text “`\nElias.\nElizabeth.`” (plus dashed-arrow back pointers).

A suffix tree has a number of useful properties that make finding repeated patterns and collecting statistics about text efficient. For example, all occurrences of any substring of the text (or simply the count of those substrings) can be retrieved in time linear only in the length of the queried substring, not the length of the input text. To find the substring s and its number of occurrences in the text, one need only traverse the suffix tree from the root along the path containing the substring.

ListReader uses Ukkonen's algorithm [58] to construct suffix trees. Therefore this step of the pipeline adds only $O(t)$ to both time and space complexity. This is true because of the following aspects of Ukkonen's algorithm: As it builds a suffix tree, it scans the text once from start to finish. As it scans, it does not store the actual suffixes in the tree; just the beginning and ending offsets as pointers into the original input string. While inserting each suffix branch into the tree, it does not increment the ending offset of any substring until it hits a mismatch and must fork. Instead, it uses a single variable that represents the current end of the text and which indicates the ending offset for all active branches in the tree simultaneously. When the algorithm encounters a new suffix that matches one already in the tree, it does no additional work except to traverse the existing branch to find how far the new suffix matches. It finds the first character in the new suffix that does not match the old branch. At that time and in that place in the tree, it creates a new fork. Since all of the suffixes of the matching branch are also somewhere in the tree and were created in a specific order, it creates back pointers (the dashed, backward-pointing edges in Figure 4.7) which are links among matching subtrees. It follows these back pointers only when it needs to update related subtrees. In this way, it can traverse and update a very limited number of subtrees that share a common prefix and a common set of suffixes.

Since ListReader conflates text before it constructs a suffix tree, all symbols in ListReader's suffix trees are conflation symbols. To illustrate a suffix tree made from conflated text, Figure 4.8 shows the suffix tree constructed from

```
[Sp][Dg].[Sp][UpLo+],[Sp][Lo].[Sp][DgDgDgDg].
[Sp][Dg].[Sp][UpLo+],[Sp][Lo].[Sp][DgDgDgDg].
[Sp]
```

which is the conflation text of the first two child records in Figure 4.1. Observe that each record has the pattern “`[Sp][Dg].[Sp][UpLo+],[Sp][Lo].[Sp][DgDgDgDg].[Sp]`”. In Figure 4.8 this pattern is embedded in (and in this case is exactly) a concatenation of the edges’ strings that label the first edge from the root to a node and the third edge emanating from that node. From the offsets, it is clear which of the nine appearances of “`[Sp]`” go with the two appearances of “`[Sp][Dg].[Sp][UpLo+],[Sp][Lo].[Sp][DgDgDgDg].[Sp]`”—offset 0 in the first edge goes with offset 1 in the second edge and offset 12 goes with offset 13. It is also clear how many repetitions of the pattern appear—the two that continue into the second edge.

4.5.4 Record Selection

In this step, ListReader searches for record patterns in the compact suffix tree constructed previously. It will do so again in a later step if it is running in the two-phase mode, which uses Steps (6) through (10). Therefore, the purpose of this step is different for one-phase and two-phase execution. In one-phase execution, the record patterns it finds here are used to construct the final regex wrapper. In two-phase execution, ListReader uses the record patterns it finds in this step to identify field groups, as explained below. Those field groups, in turn, help construct a more detailed representation of records. ListReader then uses that second set of record patterns to construct the final regex wrapper at the end of the second phase.

It is necessary to filter candidate record patterns by requiring them to be “complete”—ending in acceptable record delimiters and containing reasonable content. This is not unusual. In other grammar induction work, researchers have predefined part-of-speech patterns to constrain and filter discovered patterns to reduce errors [40]. In both one-phase and two-phase execution, ListReader selects record patterns from the conflated text in the suffix tree by searching for strings of symbols with the properties specified in ListReader’s input, whose purpose is the identification of actual records. The same kinds of constraints guide the search for candidate record patterns for both one-phase and two-phase record selection, so we now enumerate and provide intuition for the default values for these parameters. Pairs of numeric values (in parentheses, below) indicate that different parameter values are used during the first and second phases while single values indicate

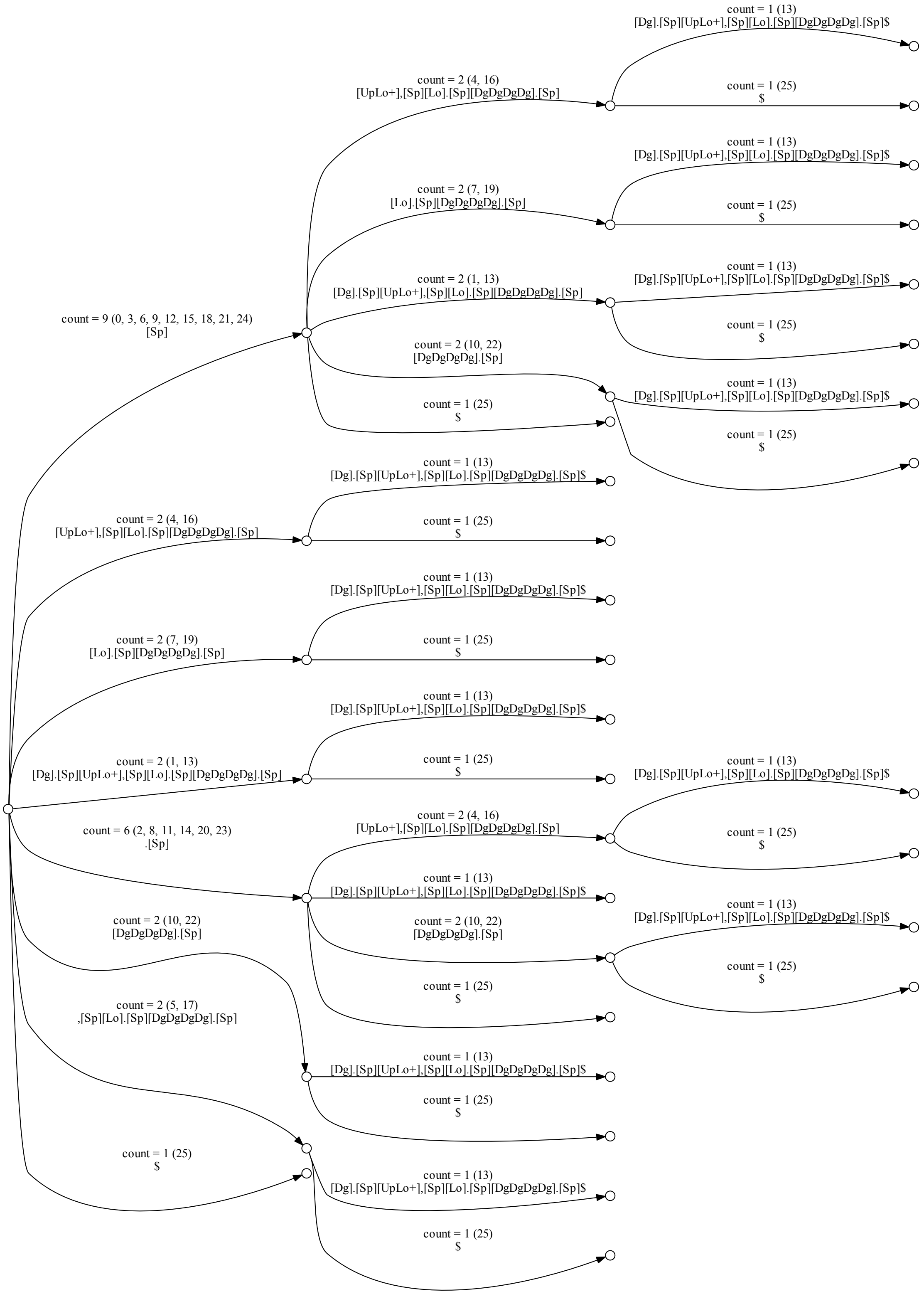

count = 1 (13)
[Dg].[Sp][UpLo+],[Sp][Lo].[Sp][DgDgDgDg].[Sp]$
count = 2 (4, 16)
[UpLo+],[Sp][Lo].[Sp][DgDgDgDg].[Sp]
count = 1 (25)
$
count = 1 (13)
[Dg].[Sp][UpLo+],[Sp][Lo].[Sp][DgDgDgDg].[Sp]$
count = 2 (7, 19)
[Lo].[Sp][DgDgDgDg].[Sp]
count = 1 (25)
$
count = 1 (13)
[Dg].[Sp][UpLo+],[Sp][Lo].[Sp][DgDgDgDg].[Sp]$
count = 2 (1, 13)
[Dg].[Sp][UpLo+],[Sp][Lo].[Sp][DgDgDgDg].[Sp]
count = 9 (0, 3, 6, 9, 12, 15, 18, 21, 24)
[Sp]
count = 1 (25)
$
count = 2 (10, 22)
[DgDgDgDg].[Sp]
count = 1 (13)
[Dg].[Sp][UpLo+],[Sp][Lo].[Sp][DgDgDgDg].[Sp]$
count = 1 (25)
$
count = 1 (25)
$
count = 1 (13)
[Dg].[Sp][UpLo+],[Sp][Lo].[Sp][DgDgDgDg].[Sp]$
count = 2 (4, 16)
[UpLo+],[Sp][Lo].[Sp][DgDgDgDg].[Sp]
count = 1 (25)
$
count = 1 (13)
[Dg].[Sp][UpLo+],[Sp][Lo].[Sp][DgDgDgDg].[Sp]$
count = 2 (7, 19)
[Lo].[Sp][DgDgDgDg].[Sp]
count = 1 (25)
$
count = 1 (13)
[Dg].[Sp][UpLo+],[Sp][Lo].[Sp][DgDgDgDg].[Sp]$
count = 2 (1, 13)
[Dg].[Sp][UpLo+],[Sp][Lo].[Sp][DgDgDgDg].[Sp]
count = 1 (25)
$
count = 1 (13)
[Dg].[Sp][UpLo+],[Sp][Lo].[Sp][DgDgDgDg].[Sp]$
count = 2 (4, 16)
[UpLo+],[Sp][Lo].[Sp][DgDgDgDg].[Sp]
count = 6 (2, 8, 11, 14, 20, 23)
.[Sp]
count = 1 (25)
$
count = 1 (13)
[Dg].[Sp][UpLo+],[Sp][Lo].[Sp][DgDgDgDg].[Sp]$
count = 2 (10, 22)
[DgDgDgDg].[Sp]
count = 1 (13)
[Dg].[Sp][UpLo+],[Sp][Lo].[Sp][DgDgDgDg].[Sp]$
count = 2 (10, 22)
[DgDgDgDg].[Sp]
count = 1 (25)
$
count = 1 (25)
$
count = 2 (5, 17)
,[Sp][Lo].[Sp][DgDgDgDg].[Sp]
count = 1 (13)
[Dg].[Sp][UpLo+],[Sp][Lo].[Sp][DgDgDgDg].[Sp]$
count = 1 (25)
$
count = 1 (25)
$
count = 1 (13)
[Dg].[Sp][UpLo+],[Sp][Lo].[Sp][DgDgDgDg].[Sp]$
count = 1 (25)
$

Figure 4.8: Suffix Tree of Conflated Text of the First Two Child Records in Figure 4.1: "[Sp][Dg].[Sp][UpLo+],[Sp][Lo].[Sp][DgDgDgDg].[Sp][Dg].[Sp][UpLo+],[Sp][Lo].[Sp][DgDgDgDg].[Sp]".

the same value is used in both. In general, we chose parameter values that ensured high precision on the development data while not reducing recall significantly. We expect that the default values of these numeric parameters will work well on most lists.

- Record patterns should begin and end with some kind of record delimiter. For phase one, the set of allowable record delimiters includes only newlines ("`\n`") by default. Future work will explore expanding the set of possible record delimiters to account for other text genres. For phase two, ListReader constructs its own symbols to represent the beginning and ending of records, "`[\n-Delim]`" and "`[\n-End-Delim]`" respectively. These symbols are used in the second phase regardless of which delimiter characters are used in the first phase.

- At least (10%) of the instances of the candidate pattern must end with the appropriate record delimiter—"`\n`" for phase-one patterns and "`[\n-End-Delim]`" for phase-two patterns. Since the suffix tree is built from conflated space symbols instead of newline characters, aligned instances of a space symbol within the suffix tree may contain a mixture of simple spaces and newlines. So it is necessary to check to see that some percentage of instances actually do contain a newline. Ideally, every record would end with the appropriate delimiter, but noise in margins in historical documents causes many records not to end cleanly. At least some should end appropriately, for if not, the pattern likely does not constitute a list record. We could have imposed a 10% constraint on the beginning delimiter instead of the ending delimiter with the same effect, except that symbols at the end of a pattern are more easily accessible within the suffix tree. To keep time complexity low—and linear with respect to t—ListReader computes the percentage from a random subset of the instances of the space symbols, which sample can be held below a maximum size that does not grow with the length of the text.

- At least (40%) of the instances of the pattern must contain both a beginning and an ending record delimiter. As with the previous heuristic, noise can cause problems, but less so when nearly half of the instances of a pattern have both a beginning and an ending delimiter. Otherwise, ListReader sometimes finds non-record patterns in which some of the instances happen to begin with a delimiter while other instances happen to end with a delimiter. If none of the instances contain both beginning and ending delimiters simultaneously, the pattern is almost certainly not a true record. ListReader does not appear to be sensitive to the exact setting of either this or the preceding parameter. Varying this parameter by 10% or so does not change the final evaluation metrics much.

- The pattern must occur at least (3) times in the input text. Record repetition constitutes a list—the more the better—but we need some minimum. In Figure 4.1 the single-name, birth-date-only record of focus in Figure 4.8 repeats six times in Figure 4.1 and likely hundreds of times in the book, but a record with three names and a spouse or two, like the Samuel Holden Parsons record in Figure 4.1, repeats much less—possibly not at all. We thus set this repetition parameter low to increase recall. Setting this parameter to at least 2 also allows ListReader to prune over half of the nodes in the suffix tree before looking for record patterns.
- The pattern must be at least (4, 2) conflated symbols long including the symbols at the beginning and ending of a record. A record should have some content. Longer patterns are more likely to be true records especially after conflation which generally makes the text less unique and patterns more ambiguous. In phase one, a symbol or two between the two record delimiters may be enough to provide sufficient content. In phase two, we encapsulate the starting record delimiter with a field group segment that includes all of the fields and delimiters between the starting record delimiter and the next field group marked with its own delimiter such as "`, b. `". Thus two symbols—the beginning and ending record segments—will contain enough meaningful content to constitute a record in phase two.
- The pattern must contain at least (1, 0) numerals or capitalized words—field-like content. In our application, and many others, field content typically contains numbers or proper nouns, which in English are capitalized, whereas common nouns and other parts of speech are not. The requirement of at least one field-like string helps increase the precision of discovered record-like patterns because it eliminates prose text that tends to contain mostly lower-case

words. Using dictionaries, this heuristic could be extended to other languages, e.g., German in which common nouns are also capitalized.

- The pattern must contain no sequence of lower-case words longer than (3) words. This heuristic improves precision just as the previous one does. Lower-case words in English list records tend to be delimiters, which are usually a sequence of just one or two words.

Record selection proceeds by finding pattern sequences that satisfy these criteria and that therefore have the expected characteristics of list records. Individual occurrences of each record pattern are already clustered and aligned within the suffix tree making it straightforward for ListReader to find candidate record patterns and identify the members of each cluster of records that share the same sequence or pattern of conflated text. To find all candidate record clusters, ListReader iterates over all the non-root internal nodes of the suffix tree whose incoming edge has the required repetition count (3 or more, as specified in the criteria above). For each node, ListReader defines a candidate record pattern as the conflated text contained in the branch between the root and the current node. From among these candidates, ListReader selects those record patterns that satisfy the remaining criteria. Because all patterns that end in the same edge also have the same number of (overlapping) instances, ListReader need only consider patterns that end in a record delimiter that is the last delimiter in its edge. This reduces the work per edge to a constant value independent of the length of the input text.

For example, letting the required repetition count be 2 (so that the example can be small enough to show), ListReader can find one candidate record pattern in Figure 4.8, namely, "`[Sp][Dg].[Sp][UpLo+],[Sp][Lo].[Sp][DgDgDgDg].[Sp]`" which it finds when considering the third child node of the node whose incoming edge has the label "`[Sp]`". This record satisfies the string-length requirement since it has 13 symbols (≥ 4). The pattern contains two numeral symbols ("`[Dg]`" and "`[DgDgDgDg]`") and one capitalized word symbol ("`[UpLo+]`"), and only one lower-case symbol ("`[Lo]`"), and thus no sequence of three or more lower-case symbols. To check record delimiters, ListReader retrieves the actual symbols at the beginnings and ends of each instance of the pattern, which for our example are at offsets 0, 12, and 24 and are all "`\n`"—a member of the set of record delimiters. No other patterns in Figure 4.8 satisfy the criteria. Either the count is less than 2 or the pattern does not begin with a record delimiter. The first child node of the node whose incoming edge has the label “`[Sp]`”, for example, has count 2, which satisfies the repetition requirement, but both occurrences of the initial “`[Sp]`” at positions 3 and 15 are space characters (“ ”) not newline characters (“`\n`”).

The first record cluster in Figure 4.9 is the actual cluster of records ListReader would produce from processing the text in Figure 4.1. Observe that the record pattern is the discovered record pattern in Figure 4.8. If ListReader creates a suffix tree for the entire text in Figure 4.1, instead of just the first two child records, it would also produce the second record cluster in Figure 4.9. With a larger input text, ListReader could potentially identify all six record types present in Figure 4.1.

- `[Sp][Dg].[Sp][UpLo+],[Sp][Lo].[Sp][DgDgDgDg].[Sp]`
  - “`\n1. Andrew, b. 1772.\n`”
  - “`\n2. Clarissa, b. 1774.\n`”
  - “`\n3. Elias, b. 1776.\n`”
  - “`\n5. PoUy , b. 1782.\n`”
  - “`\n5. Sylvester, b. 1782.\n`”
  - “`\n7. Charles, b. 1787.\n`”
  - “`\n8. Margaret Stoutenburgh, b. 1794.\n`”
- `[Sp][Dg].[Sp][UpLo+],[Sp][Lo].[Sp][DgDgDgDg],[Sp][Lo].[Sp][DgDgDgDg].[Sp]`
  - “`\n4. William Lee, b. 1779, d. 1802.\n`”
  - “`\n6. Nathaniel Griswold, b. 1784, d. 1785.\n`”
  - “`\n3. Lucia, b. 1777, d. 1778.\n`”
  - “`\n6. Phebe, b. 1783, d. 1805.\n`”
  - “`\n7. William Richard Henry, b. 1787, d. 1796.\n`”

Figure 4.9: Record Clusters from Figure 4.1.

Given an identified record in a suffix tree, ListReader completes its induction of the grammar for the induced wrapper by adding the root non-terminal “`[Record]`” to the partially created parse tree in the conflation step as Figure 4.10 shows for the first record in Figure 4.6. Note that the dashed arrows, which are not part of the parse tree, give the sequence of symbols of the identified pattern that defines the record cluster in Figure 4.8.

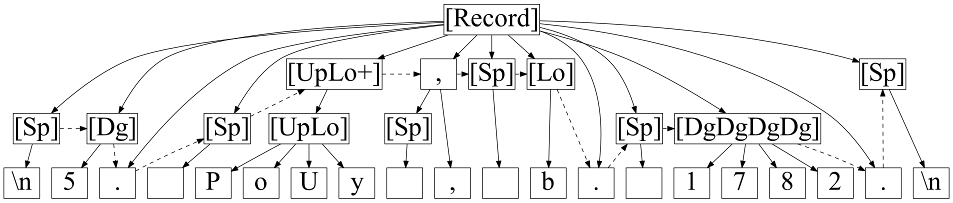


Figure 4.10: Phase-one Parse Tree of the Text “`\n5. PoUy , b. 1782.\n`”.

Since a suffix tree for text of length t has at most t internal nodes, this step adds $O(t)$ time and space because it iterates over all non-root internal nodes of the suffix tree whose incoming edge has the appropriate occurrence count. As it iterates, it does a constant amount of work to check conformance with the record selection criteria and form the record clusters. The number of record clusters does not grow with the length of the input text but is constant with a fixed maximum pattern length (about the length of a page).

4.5.5 Record Cluster Adjustment

The purpose of creating record clusters is to facilitate the labeling of fields so that ListReader can extract the field values and map them to an ontology. The labeling of one record in a cluster is sufficient to label them all because they all satisfy the same record pattern. Unfortunately, it is possible for a substring of the document text to be in more than one cluster and thus be labeled more than once. The substring may even be labeled in different (and therefore in incorrect) ways. ListReader can, and does, avoid multiple labelings of a string in the active sampling step below by marking text that has been labeled and rejecting a subsequent attempt to label the text. However, ListReader can better avoid this issue and improve the quality of the record clusters by making adjustments to clusters as soon as they are formed.

To motivate the adjustments, consider the pattern

```
"[Sp][Dg].[Sp][UpLo+],[Sp][Lo].[Sp][DgDgDgDg].[Sp]
     [Dg].[Sp][UpLo+],[Sp][Lo].[Sp][DgDgDgDg].[Sp]"
```

which subsumes the pattern

```
"[Sp][Dg].[Sp][UpLo+],[Sp][Lo].[Sp][DgDgDgDg].[Sp]".
```

These two patterns and their record clusters would have been created from a suffix tree for the full text in Figure 4.1. Observe in Figure 4.8 that these two patterns are in the path from the `[Sp]` edge to the two edges beginning with `[Dg]`. The edge counts in this figure are not high enough to select the longer pattern, but they would be in a suffix tree built from the entire text because the third child record in Figure 4.1 would also have been included. The record cluster selected from the longer pattern is in Figure 4.11, whereas the record cluster selected from the shorter pattern is the first cluster in Figure 4.9. Now, observe that the substring “2. Clarissa, b. 1774.” appears in both clusters and also twice in the cluster in Figure 4.11 and that the substrings “1. Andrew, b. 1772.” and “3. Elias, b. 1776.” appear in both clusters. Ideally, the user should be asked to label only one member of the shorter child record pattern.

```
1. Andrew, b. 1772. 2.Clarissa, b. 1774.
2. Clarissa, b. 1774. 3. Elias, b. 1776.
```

Figure 4.11: Record Cluster for Two-Child Records.

When one record pattern subsumes another, ListReader assigns record strings to only one of the two patterns. It scores and ranks each candidate by the product of the pattern’s length and frequency (occurrence count) and assigns it to the highest-ranking pattern.[4] For our example, since the pattern length for the first record cluster in Figure 4.9 is 13 and it initially has 7 records, its score is 91 which ranks higher than 50 (= 25×2), the score for the longer pattern in Figure 4.11. Thus, in this case, ListReader assigns all three candidate child-record substrings in Figure 4.11 to the first cluster in Figure 4.9 and discards the two-record cluster. This is good for two reasons. The records

[4] Work on information compression justifies our choice of scoring and ranking [55]. See Appendix B for a full explanation.

end up in the intuitively best cluster and we need not process the discarded cluster. ListReader always removes full record strings from a list, including when only a proper substring of a record overlaps with a higher-scoring pattern.

This step adds $O(t)$ time because it iterates over the instances of each record cluster produced in the previous step. While the number of instances of each cluster is $O(t)$, the size of the set of accepted and stored record clusters is a constant that does not depend on the length of the input text, as it is the length of each record instance. This step does not add to the space complexity as it removes records from clusters.

4.5.6 Field Group Delimiter Selection

Phase two execution, Steps (6)–(10), makes additional adjustments to the structure of records to further reduce the cost of labeling. Observe in Figure 4.9 that the three fields (*ChildNr*, *GivenName*, and *BirthDate.Year*) in the first record cluster are also the first three fields in the second record cluster. If ListReader can confidently determine that these fields should be labeled the same, labeling these three fields in a member of either cluster is sufficient to allow ListReader to automatically label these fields in both clusters and thus reduces the amount of required human labeling. Therefore, ListReader identifies what we call *field group delimiters* such as “`, b. `” or “`, d. `” which respectively precede birth years and death years in Ely child records and allow ListReader to confidently align field groups across record clusters.

ListReader creates a set of generic delimiters from the record clusters it produces in the previous step. Each generic delimiter contains, and is identified by, the string of lower-case words that appear as identical substrings at a fixed position within four or more record clusters. In Figure 4.9, for example, “`b`” appears at a fixed position in both clusters, and “`d`” appears at a fixed position in the second cluster. ListReader expands each generic delimiter into a set of longer, more specific delimiters that differ from each other by surrounding space and punctuation characters. For example, a generic “`b`” delimiter might have two specific types: “`, b. `” and “`; b. `”. Each specific delimiter must appear in at least two record clusters to be selected. ListReader also treats record delimiters as field group delimiters. For the clusters in Figure 4.9, therefore, the identified field group delimiters would be "\n" (beginning record delimiter), ", b. " (birth event delimiter), ", d. " (death event delimiter), and ".\n" (ending record delimiter).

This step adds $O(t)$ time and space because it iterates over all members of the record clusters. It records a constant number of generic delimiters containing pointers to the delimiter instances. If ListReader cannot identify any field group delimiters in the text beyond the record delimiters, the two-phase ListReader pipeline reduces to a one-phase pipeline, and Steps (6)–(10) are skipped.

4.5.7 Field Group Template Construction

ListReader creates a field group template for each discovered field group delimiter, including the starting but not ending record delimiters. It creates a field group template of type T (e.g., type "b" containing birth event information) from: (1) the union of the specific delimiters of type T followed by (2) the union of the conflated text between each instance of the field group delimiter of type T and the following occurrence of any other field group delimiter in the same record. These field group templates represent a more general set of text than the text from which ListReader generated them because they represent any combination of delimiter text and field group text of the same type, including combinations not found within clustered records.

For example, suppose the pipeline thus far had discovered two record clusters containing marriage information in the format of ", m. 1801 Edward Hill" and "; m. John Marvin". In this case, ListReader would have produced an "m" field template like the following which contains two delimiter variations and two content variations:

"[[; m.] | [, m.]] [[[UpLo+]] | [[DgDgDgDg][Sp][UpLo+]]]".

Then, even though the text string "; m. 1771, Lucinda Lee" may not appear in the clustered records, the generated field group template for marriage data would be able to recognize it since it is a combination of field groups and their delimiters it has seen.

The identification of field group templates allows ListReader to align and cluster these field group segments—segments of text that include a field group delimiter and its associated field group(s). This is the main distinction between the one-phase and two-phase variations of the grammar induction pipeline. The significance is that field group clusters are generally larger (contain more members or occurrences) than clusters of whole records because field group segments are smaller constituents of records and have less opportunity for variations in fields that divide the clusters. Aligning the more numerous and smaller field groups within records reduces labeling cost. For example, for the record clusters in Figure 4.9, ListReader would produce three field group templates: a starting record template “`[\n] [[Dg].[Sp][UpLo+]]`”, a birth-year template “`[, b. ] [[DgDgDgDg]]`”, and a death-year template “`[, d. ] [[DgDgDgDg]]`”. Then, the labeling of these templates only needs to be done once, rather than once for each record cluster in which they appear.

This step adds $O(t)$ time and $O(1)$ space because it iterates over all the occurrences of field group delimiters and contents and stores only a limited number of templates.

4.5.8 Field Group Parsing

This step extends the conflation parsing that began in Step 2 (Subsection 4.5.2). Unlike that step, the patterns being matched are the field group templates created in Step 7 (Subsection 4.5.7). The conflation symbols, appropriately, are “`[\n-Segment]`” and “`[\n-End-Segment]`” for the beginning and ending record delimiters and “`[`T`-Segment]`” for field groups of type T. For example, ListReader, having already replaced the text

“`\n7. Charles, b. 1787.\n8. Margaret Stoutenburgh, b. 1794.\n`”

with

“`[Sp][Dg].[Sp][UpLo+],[Sp][Lo].[Sp][DgDgDgDg].[Sp][Dg].[Sp][UpLo+], [Sp][Lo].[Sp][DgDgDgDg].[Sp]`”

would now replace it again with a new sequence:

“`[\n-Segment][b-Segment][\n-End-Segment][\n-Segment][b-Segment][\n-End-Segment]`”

Note that the newline character between these two records becomes a constituent of two adjacent symbols: “`[\n-End-Segment]`” and “`[\n-Segment]`”.

The parsing algorithm consists mainly of a linear scan over the input text testing for occurrences of a fixed number of field group templates. It first searches for the first lowercase word of each delimiter. For each limited match, it checks for the rest of the delimiter and then the field group content. For complete segment matches, it replaces the matching strings with appropriate *Segment* symbols and their two constituents (delimiter and field group). This step therefore contributes $O(t)$ time and space to the pipeline.

4.5.9 Suffix Tree Construction (2)

ListReader constructs a new suffix tree from text conflated in the previous step using field group templates. As a small example, consider the text:

```
Children;
1. John
2. Mary
```

and the corresponding conflated text sequence

```
"[UpLo+];[Sp][Dg].[Sp][UpLo+][Sp][Dg].[Sp][UpLo+][Sp]".
```

Step 8 (Section 4.5.8) would transform that text into

```
"[UpLo+];[\n-Segment][\n-End-Segment][\n-Segment][\n-End-Segment]"
```

where “`[\n-Segment]`” has as one of its alternatives the pattern “`[\n] [[Dg].[Sp][UpLo+]]`”. From this conflation sequence, the current step (Step 9) would produce the suffix tree in Figure 4.12. The two target records appear in the first branch descending from the root. This time, the whole record pattern with two occurrences appears in a single edge of this smaller suffix tree, where it can be found in the next step.

This step contributes $O(t)$ time and space to the pipeline just as the first suffix tree construction step did.

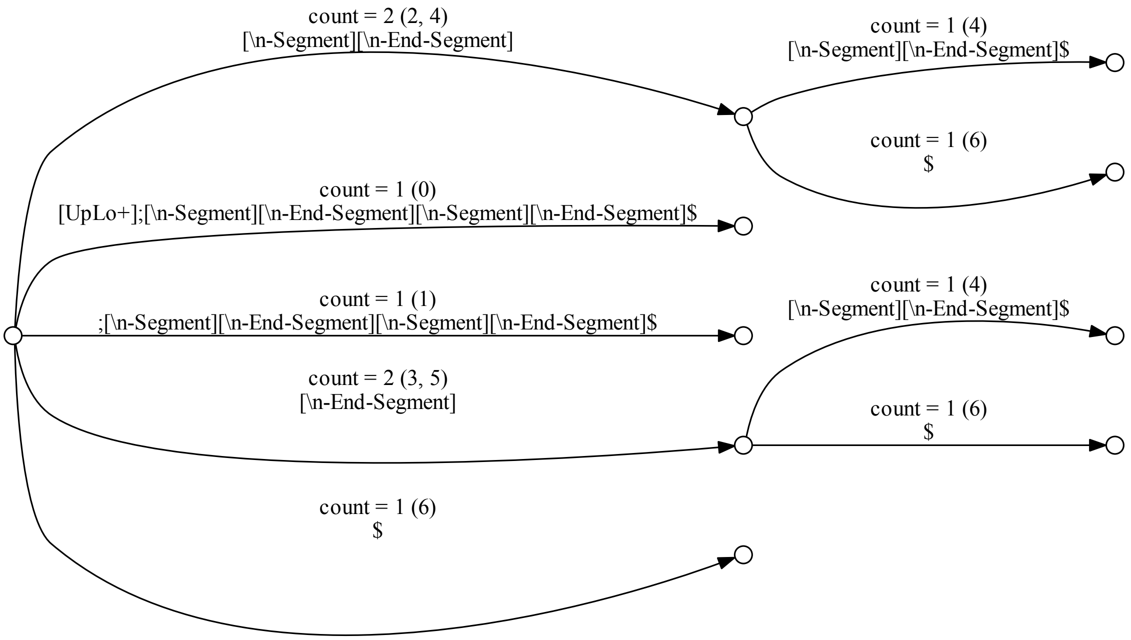

count = 2 (2, 4)
[\n-Segment][\n-End-Segment]
count = 1 (4)
[\n-Segment][\n-End-Segment]$
count = 1 (6)
$
count = 1 (0)
[UpLo+];[\n-Segment][\n-End-Segment][\n-Segment][\n-End-Segment]$
count = 1 (4)
[\n-Segment][\n-End-Segment]$
count = 1 (1)
;[\n-Segment][\n-End-Segment][\n-Segment][\n-End-Segment]$
count = 2 (3, 5)
[\n-End-Segment]
count = 1 (6)
$
count = 1 (6)
$

Figure 4.12: Suffix Tree of “`[UpLo+];[\n-Segment][\n-End-Segment][\n-Segment][\n-End-Segment]`”.

4.5.10 Revised Record Selection

ListReader selects a new set of record candidates and forms new record clusters from the suffix tree as explained in the first record-selection step (Subsection 4.5.4). For example, if the occurrence count constraint were set to two, ListReader would select the pattern “`[\n-Segment][\n-End-Segment]`” from the suffix tree in Figure 4.12 because it is two symbols long, contains no long sequences of lower-case words, and 100% of the occurrences of the pattern contain both a beginning and ending record delimiter.

This step is made more robust to errors in earlier stages of the pipeline by allowing additional record patterns to be found without losing the patterns found in the first record-selection step. For example, if ListReader had failed to find any field group delimiters in previous steps and therefore was unable to produce the pattern “`[\n-Segment][\n-End-Segment]`”, it would still discover the pattern “`[Sp][Dg].[Sp][UpLo+][Sp]`” in the first parse tree.

As is the case for phase-one record selection, ListReader induces a parse tree for each record. Figure 4.13 shows the parse tree constructed in phase-two from the text of the first record in Figure 4.6, “`\n5. PoUy , b. 1782.\n`”. For each field group segment (except the end-record delimiter), ListReader adds “[T-Delim]” for the delimiter constituent of the segment and “[T-FieldGroup]” for the field group constituent, where T is the field group type. For the end delimiter, ListReader adds “`[\n-End-Segment]`”. Finally, ListReader adds “`[Record]`” as the root of the parse tree.

Like the phase-one record selection step, ListReader iterates over all non-root internal nodes of the suffix tree whose incoming edge has the appropriate occurrence count and does a constant amount of work at each node. This step usually considers fewer nodes than in phase one. Still, this step also contributes $O(t)$ time and space to the pipeline.

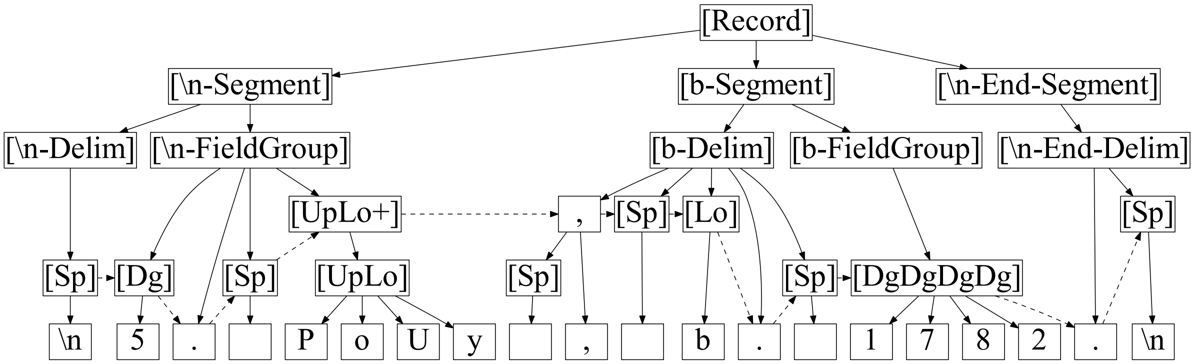


Figure 4.13: Phase-two Parse Tree of the Text “`\n5. PoUy , b. 1782.\n`”.

4.5.11 Regex Construction

ListReader creates a regular expression from the set of induced parse trees for each record cluster (e.g., Figures 4.10 and 4.13). The resulting regex is an alternation of all the regular expressions, one for each record cluster. For a parse tree, ListReader creates a regular expression by surrounding sub-expressions with parentheses to either group alternatives for a given node in the parse tree or to mark capture groups—any node in the parse tree that should receive a unique label during active sampling. Figure 4.14 gives two regular expressions, which we use as examples as we describe regex construction: (a) for the phase-one parse tree in Figure 4.10 and (b) for the phase-two parse tree in Figure 4.13.

```
([\n])([\d]{1})(\.)([ \n])([A-Z]+[a-z]+|[A-Z]+[a-z]+[A-Z]+[a-z]+)
    (( )?,)([ \n])([a-z]+)(\.)([ \n])([\d]{4})(\.)([\n])
```

(a)

```
(([\n])([\d]{1})(\.[ \n])(([A-Z]+[a-z]+|[A-Z]+[a-z]+[A-Z]+[a-z]+)))
    ((( )?,)([ \n])(b)(\.)([ \n])([\d]{4}))((\.)([\n]))
```

(b)

Figure 4.14: Regular Expressions for the Parse Trees in (a) Figure 4.10 and (b) Figure 4.13.

The smallest parenthesis-enclosed expressions are for the word-level conflation symbols (e.g., "`[Sp]`", "`[Lo]`", "`[UpLo]`", "`[Dg]`", and "`[DgDgDgDg]`"). Notice in Figure 4.14 that the `[Sp]` symbols beginning and ending the record pattern are constrained to be record delimiters, "`\n`", while the internal spaces can be either "`\n`" or " ", which greatly improves the precision of the regex while still allowing for multiline records to be matched. Digit sequence symbols of length n become "`[\d]{`n`}`". Since the `[UpLo+]` symbol conflates a variable-length sequence of capitalized words, ListReader generates an alternation of one or two or ... n, parenthesized, space-separated sequences of "`([A-Z]+[a-z]+|[A-Z]+[a-z]+[A-Z]+[a-z]+)`", which accommodates up to an n-word sequence, where n is the longest word sequence expected or observed. (In Figure 4.14, we have given only the first element of the sequence, because it both fits the example text and avoids unnecessary clutter.) For the special case of a space preceding punctuation, "`( )?`" precedes the punctuation mark, allowing for one extra space. Sequences of one or more lower-case letters, `[Lo]`, become "`[a-z]+`". For phase-two parse trees, ListReader also adds parentheses for any alternation group, e.g., for the name alternation in Figure 4.14b, and for each segment variation.

In this step, ListReader also initializes an array of capture group labels. Each label corresponds to a capture group (matching pair of parentheses) in the regex and is either "do not label", "record delimiter", or an integer. ListReader associates "do not label" with all of the larger capture groups that contain smaller capture groups. ListReader assigns "record delimiter" to the "`([\n])`" capture groups that begin and end each record template expression to group fields that belong to the same primary object during final extraction. ListReader initializes the rest of the capture group labels to integer values so as to minimize active sampling cost as explained next.

To minimize labeling effort during active sampling, ListReader recognizes equivalent fields across record clusters, which are then labeled only once, independent of the number of different record clusters in which they appear. Fields are equivalent if they have the same content and context. In particular, two fields' capture groups are equivalent if they (1) are of the same conflation type (as determined in the first conflation step), (2) appear in the same type of field group (e.g., "`b`" vs. "`\n`"), and (3) are surrounded by text within their field groups that all have identical conflation types (also as determined in the first conflation step). These constraints define a set of equivalence classes which ListReader then distinguishes, labeling all fields across record clusters with the same identifying integer that belong to the same equivalence class. Using these identifying integer labels (these IDs), ListReader can later assign labels to all capture groups that should be the same after the user labels any one of them. For example, ListReader would assign the same IDs to the child number, name, and birth-year capture groups in the record templates of "`\n5. PoUy, b. 1782.\n`" and "`\n6. Phebe, b. 1783, d. 1805.\n`" despite being in different record clusters.

In forming these equivalence classes for labeling, ListReader must be careful not to be overly aggressive. Because the equivalence-class-creation rules are conservative, sometimes fields that eventually do have the same label have to be labeled separately by the user. For example, in the record templates for the two records

"`\n2. Elizabeth, b. 1774, d. 1851, m. 1801 Edward Hill`"

and

"`\n4. Lucia Mather, b. 1779, d. 1870, m. John Marvin`"

all four names would initially have different IDs even though in the end they may all be labeled the same. The names "`Edward Hill`" and "`John Marvin`" in the marriage segments are assigned different IDs because their field group templates differ: one contains a marriage year and the other does not.[5] The names "Elizabeth" and "Lucia Mather" are assigned different IDs because they have different conflation-symbol lengths (one symbol versus two symbols), and ListReader has no justification to assign the same label to these two names without input from the user. One user may assign "Full Name" to both whole names (including the space between "Lucia" and "Mather") while another user may assign finer-grained labels such as "Given Name" and "Middle Name" to the individual parts. Even two-word names may be labeled differently: "Lucia Mather" may be labeled "First Given Name"/"Second Given Name" while both "Edward Hill" and "John Marvin" may be labeled "Spouse Name". Note, however, that the rest of the corresponding fields in this example (i.e. the child numbers, birth years, and death years) will share IDs and will require only one of the templates to be labeled to label both.

Regex creation consists in iterating over all the record templates. For each record template, ListReader recursively traverses the finite, constant-depth syntax tree to generate each piece of the regex. Each field group segment has at most $O(t)$ alternatives. Duplicate field group templates may be generated as part of different record templates but are then immediately discarded when ListReader discovers that they are duplicates (and should contain the same capture group IDs). This step therefore adds $O(t)$ time and $O(1)$ space to the pipeline.

4.5.12 Active Sampling

The active sampling step consists of a cycle of repeated interaction with the user who labels the fields in the text of a record from some template that ListReader selects. Actual labeling consists of copying substrings of the ListReader-selected text into the entry fields of a form. The structure of the form and the names of the form fields constitute the label as explained in Sections 4.3 and 4.4. On each iteration of the loop, the user updates the form, if necessary, and labels the ListReader-chosen and ListReader-highlighted text. ListReader then accepts the labeled text via the Web form interface and assigns labels to the corresponding capture groups of the regex wrapper.

[5] Here, ListReader is likely being too conservative. Loosening equivalence requirements is possible under specific assumptions and would likely improve recall significantly. We intend to investigate this in future work.

Most approaches to active learning have two key steps: active sampling and model update, with the active sampling step being the hallmark of active learning [53]. This ListReader step is an *active sampling* process and not a full *active learning* process because it does not update the already-learned model (the regex in our case). It does, however, re-label some of the ListReader-generated labels to enable the mapping of extracted information to the ontology. In each cycle, ListReader actively selects the text for labeling that maximizes the return for the labeling effort expended. Our approach is thus like other *unsupervised active learning* approaches [38] that do not update the already-learned model. Indeed, the regex learning ListReader does is fully unsupervised—no regex learning takes place under the supervision of a user either interactively or in advance. On the other hand, we cannot say that the whole ListReader process is unsupervised because to do so would ignore the value of the labels the user does provide. Producing a mapping from capture group numbers to ontology predicates without supervision is not trivial.

Hu et al. ([38]) suggest three benefits of unsupervised active learning compared to supervised active learning. First, looking for natural clusters within a feature space in an unsupervised manner before any labels are provided prevents the system from incorrectly conflating samples in that space that may share the same label but are distinct in features. Second, supervised active learning may sometimes fail to select new samples that belong to new categories (i.e. have an unknown label) because those samples usually lie far from decision boundaries. Third, it is much easier to adapt a model learned through unsupervised active learning than one learned through supervised active learning to a new target schema because only the labels need to change, not the rest of the model. These benefits are also true of our approach.

To initialize the active sampling cycle, ListReader applies the regex to the text of each page in the book. It labels the strings that match each capture group with the capture group's label, which is initially just a number as explained in Subsection 4.5.11. ListReader then saves the count of matching strings for each capture group integer. It also records the page and character offsets of the matching strings throughout the book and associated integers. ListReader then initializes the first active sampling cycle by querying the user for the labels of the "best" string by displaying the appropriate page and highlighting the string.

The string ListReader selects as "best" is a string that matches the sub-regex of the ListReader-generated regex with the highest predicted return on investment (ROI), where the selected sub-regex corresponds to a single record cluster and is either one of, or part of one of, the top-level alternations of the generated regex. When more than one such string is in the document, ListReader selects the first one on the page containing the most matches of the sub-regex. One can think of ROI as the slope of the learning curve: higher accuracy and lower cost produce higher ROI. ListReader computes predicted ROI as the sum of the counts of the strings matching each capture group in the candidate sub-regex divided by the number of capture groups in the sub-regex. It limits the set of candidate sub-regexes to those that are contiguous and complete, meaning sub-regexes that contain no record delimiters or previously-labeled capture groups and that are not contained by any longer candidate sub-regex. Querying the user to maximizing the immediate ROI tends to maximize the slope of the learning curve and has proven effective in other active learning situations such as [33]. In preliminary experiments, we found this query policy to improve our final evaluation metrics more than a policy based only on highest match counts (without normalizing by sub-regex length). From our example page (Figure 4.1), the string ListReader would select for manual labeling is "`\n1. Andrew, b. 1772.\n`". The single digit child number and associated delimiter text ("`[Dg].[Sp]`") occur 15 times in the context of "`\n[Dg].[Sp][UpLo+]`" (its field group template). The single given name ("`[UpLo]`") occurs 9 times in the same context. The second field group pattern ("`, b. [DgDgDgDg]`") occurs 17 times among all the records of the page, and whose ending period occurs 15 times next to a newline. Since the pattern has 11 unlabeled capture groups and therefore a predicted cost of 11, the predicted ROI is 15.5 = $(15 \times 3 + 9 \times 1 + 17 \times 6 + 15 \times 1)/11$. This is higher for the example page than the predicted ROI of any other sub-regex. To label this text, the user would copy "1" into the *ChildNr* field of the form in Figure 4.2, copy "Andrew" into the first entry blank in the *GivenName* field, and copy “1772” into the *BirthDate.Year* field, making an actual cost of three user-specified labels. (The rest of the fields in the form would be empty).

Once labeling of the selected text is complete, ListReader removes the counts for all strings that match the corresponding capture groups, recomputes the ROI scores of remaining capture groups, and issues a query to the user. For the page in Figure 4.1, in the second active-sampling cycle, ListReader would highlight “`, d. 1802`” in the record “`4. William Lee, b. 1779, d. 1802`”, clear the form, and add the already-labeled ”`4`” and ”`1779`” in the *ChildNr* and *BirthDate.Year* fields. The user would then place “1802” in the *DeathDate.Year* field.

The number of iterations of active sampling depends on the budget determined by the user—how many field labels the user is willing to label. ListReader can give the user help in deciding how long to work by displaying the number of times the current pattern matches text and therefore how many additional fields would be labeled. The user must decide how long to provide labels. The longer the user works, the less text is extracted by each additional label. However, it should be noted that even if the user continues to the end when manual labeling applies to only the text the user is labeling, ListReader is still saving the user time by automatically finding the text to label.

Because of the way ListReader produces the initial grammar, including the gathering of statistics about records and fields, active sampling is impactful from the very first query. Compared with typical active learning [53], ListReader need not induce an intermediate model from labeled data before it can become effective at issuing queries. Furthermore, ListReader need not know all the labels at the time of a query. Indeed, it starts active sampling without knowing any labels. The query policy is similar to processes of novelty detection [45] in that it identifies new structures for which a label is most likely unknown. Furthermore, the grammar can be induced for complete records regardless of how much the user annotates or wants extracted, and ListReader is not dependent on the user to identify record- or field-delimiters nor to label any field the user does not want to be extracted.

Active sampling initialization adds $O(t)$ to both time and space complexity because ListReader must apply the regex to each page of text and record the location of each string that matches each capture group. Regular expressions can be executed in $O(t)$ time. Each iteration of the active sampling cycle adds nothing in terms of space complexity—it merely changes the labels of capture groups. Each loop does add $O(t)$ to the time complexity because it must find and update a number of occurrences of the labeled pattern that is proportional to the size of the document in the worst case. The number of iterations of active sampling is also a function of the size of the label alphabet, therefore, unlike previous steps, this step is also linear in the number of output labels.

4.5.13 Wrapper Output

Having constructed the regex wrapper, ListReader applies the regex with its final array of capture group labels, translates the labeled text into predicates as explained in Section 4.4, and inserts them into the ontology. Any remaining unlabeled text produces no output. Given the two record clusters of Figure 4.9 and the ontology of Figure 4.3, if the user provides the 9 field labels that would be requested for these clusters after phase-two processing, for example, ListReader would extract predicates for all 12 records including 75 binary predicates (relationships) and 87 unary predicates (objects connected in relationships, including the 12 *Person* objects). Because of cross template labeling for field groups, providing these same 9 labels is sufficient to capture from the information in Figure 4.1: 17 of the 19 birth years (b-fields) including 2 in the family-header records,[6] 10 of the 11 death years (d-fields) including the 2 in the family-header records,[7] all 15 child numbers, and all 23 names in the child records (but not names in the family-header records, for those would need to be labeled separately)—assuming, of course, that the record templates for the rest of the parent and child records are discovered when processing the whole book. From the entire Ely Ancestry book, providing these same 9 labels is sufficient to capture over 3,800 birth years, 900 death years, 4,400 child numbers, and 7,500 names and produce over 40,000 predicate assertions. The field labels and extracted data in this example are a subset of the data involved in the evaluation, below.

[6] The two birth years not included are in “who-was-b” fields rather than “b” fields.

[7] The one death year not included is in an “and-d” field rather than a “d” field.

This step adds $O(t)$ to both time and space complexity in terms of the input text length and $O(f)$ to both time and space in terms of the size of the field label alphabet because it adds $O(1)$ number of predicates to the ontology for each additional field label.

To conclude the discussion of time and space complexity for the entire pipeline, we note that since all steps are sequenced within a pipeline architecture, we take the largest single step to determine the overall time complexity, which is $O(t)$, or linear in the size of the input text. The overall space complexity is the sum of the individual steps' space complexity in the worst case, and is therefore also $O(t)$. Similarly, the entire pipeline is also linear in terms of label alphabet size.

4.6 Evaluation

In this section we describe the data (books) we used to evaluate ListReader. We explain the experimental procedure in which we compared ListReader's performance with the performance of an implementation of the conditional random field (CRF) as a comparison system. We give the metrics we used and the results of the evaluation, which includes a statistically significant improvement in F-measure as a function of labeling cost.

4.6.1 Data

General wrapper induction for lists in noisy OCR text is a novel application with no standard evaluation data available and no directly comparable approaches other than our own previous work. Therefore, we produced development and final evaluation data for the current research from two separate family history books.[8]

We developed the ListReader system using the text of *The Ely Ancestry* [6]. *The Ely Ancestry* contains 830 pages and 572,645 word tokens.[9] We completed all design, implementation, and parameter tuning for ListReader using *The Ely Ancestry* before looking for, selecting, or hand labeling the book on which we would perform final evaluations to avoid biasing the implementation

[8] We will make all text and annotations available to others upon request.

[9] Appendix A contains three sample pages.

with knowledge of the test data. We selected *Shaver-Dougherty Genealogy* [54] as our final evaluation text. *Shaver-Dougherty Genealogy* contains 498 pages and 468,919 words.[10] We selected this book randomly from a subset of the 100,000+ family history books being collected at FamilySearch.org. We produced the initial subset of family history books in three steps: First, we automatically removed books with low genealogy-data content based on a third-party tool that looks for genealogy-related words, names, dates, etc. Second, we manually inspected several pages from each book and removed those books whose OCR contained obvious problems, primarily zoning errors where the page text was not rendered in true reading-order (e.g., interleaving two columns of text). Third, we kept only books that contained at least two kinds of lists, e.g., an index list at the back of the book and the typical family lists in the body of the book.

To label training, development, and test data, we built a form in the ListReader web interface that contained all the information about a person visible in the lists of selected pages. Using the tool, we selected and labeled complete pages from the *Shaver-Dougherty Genealogy* book. The web form tool generated and populated the corresponding ontology which we used as test data for ListReader and training and test data for the comparison CRF. For final evaluation data, we hand labeled enough pages from *Shaver-Dougherty Genealogy* for ListReader's active sampling to reach the cost of 90 hand-labeled fields for both one-phase and two-phase execution, plus 25 randomly-selected pages to ensure we had representatives from all parts of the book, especially prose pages. Prose pages provide non-list text (negative training examples) which is necessary to train the CRF to discriminate between list-text and non-list-text. Furthermore, the 25 random pages also allow a more complete evaluation of both systems in terms of checking for false positives that might occur if certain instances of prose text happened to appear similar to the patterns learned from lists. In all, we annotated 68 pages of *Shaver-Dougherty Genealogy*. The annotated text from the 68 pages have the following statistics: 14,314 labeled word tokens, 13,748 labeled field instances, 2,516 record instances, and 46 field types. Table 4.2 shows the 46 field labels.

[10] Appendix A contains three sample pages.

Table 4.2: Field Type Labels.

Ancestor[1].BirthOrder	Marriage[1].Date.Day
Ancestor[2].BirthOrder	Marriage[1].Date.Month
Ancestor[3].BirthOrder	Marriage[1].Date.Year
Ancestor[4].BirthOrder	Marriage[1].Place.County
Ancestor[5].BirthOrder	Marriage[2].Date.Day
Ancestor[6].BirthOrder	Marriage[2].Date.Month
Ancestor[7].BirthOrder	Marriage[2].Date.Year
Ancestor[8].BirthOrder	Marriage[2].Place.County
Ancestor[9].BirthOrder	Name.GenSuffix
Birth.Date.Day	Name.GivenName[1]
Birth.Date.Month	Name.GivenName[2]
Birth.Date.Year	Name.GivenName[3]
Birth.Place.City	Name.Surname
Birth.Place.County	PageNumber[1]
Birth.Place.State	PageNumber[2]
BirthOrder	PageNumber[3]
ChildCount	PageNumber[4]
Death.Date.Day	Spouse[1].Name.GivenName[1]
Death.Date.Month	Spouse[1].Name.GivenName[2]
Death.Date.Year	Spouse[1].Name.Surname
Death.Place.City	Spouse[2].Name.GivenName[1]
Death.Place.County	Spouse[2].Name.GivenName[2]
Death.OtherInfo	Spouse[2].Name.Surname

4.6.2 CRF Comparison System

Since general wrapper induction for lists in noisy OCR text is a novel application with no standard baseline, we wish to give the reader a sense of the difficulty of this application in familiar terms. The Conditional Random Field (CRF) is a general approach to sequence labeling, achieving state-of-the-art performance in a number of applications. Being a highly developed statistical approach, it should do well at weighing evidence from a variety of features to robustly extract fielded information in the face of random OCR errors, ambiguous delimiters, and other challenges in this kind of text. We believe the performance of the supervised CRF serves as a good baseline or reference point for interpreting the performance of ListReader. The CRF implementation we applied is from the Mallet library [46].

Despite our goal of eliminating knowledge engineering from the cost of wrapper induction, we went through a process of feature engineering and hyper-parameter tuning for the CRF to further ensure a strong baseline. The feature engineering included selecting an appropriate set of word token features that allowed the CRF to perform well on the development test set. The features we applied to each word are listed in Table 4.3. The dictionaries are large and have good coverage. We also distributed the full set of word features to the immediate left and right neighbors of each word token (after appending a "left neighbor" or "right neighbor" designation to the feature value) to provide the CRF with contextual clues. (Using a larger neighbor window than just right and left neighbor did not improve its performance.) These features constitute a much greater amount of knowledge engineering than we allow for ListReader.

Table 4.3: CRF Word Token Features.

Case-sensitive text of the word
Dictionary/regex Boolean flags:
Given name dictionary (8,428 instances)
Surname dictionary (142,030 instances)
Names of months (25 variations)
Numeral regular expression
Roman numeral regular expression
Name initial regular expression (a capital letter followed by a period)

We simulated active learning of a CRF using a random sampling strategy. Random sampling is still considered a hard baseline to beat in active learning research, especially early in the learning process when learner exploration is a more valuable sample strategy than exploitation of the trained model [14]. Our aim of low cost motivates us to focus on the early end of the learning curve.

Each time we executed the CRF, we trained it on a random sample of n lines of text sampled throughout the hand-labeled portion of the corpus. Then we executed the trained CRF on all remaining hand-labeled text. We varied the value of n from 1 to 20 to fill in a complete learning curve. We ran the CRF a total of 7,300 times and then computed the average y value (precision, recall, or F-measure) for each x value (cost) along the learning curve.

4.6.3 Experimental Procedure and Metrics

To test the extractors, ListReader and the CRF, we wrote an evaluation system that automatically executes active sampling by each extractor, simulates manual labeling, and completes the active sampling cycle by altering labels for ListReader and by retraining and re-executing the CRF. The extractors incur costs during the labeling phase of each evaluation run which includes all active sampling cycles up to a predetermined budget. To simulate active sampling, the evaluation system takes a query from the extractor and the manually annotated portion of the corpus and then returns just the labels for the text specified by the query in the same way the ListReader user interface would have. In this way, we were able to easily simulate many active sampling cycles within many evaluation runs for each extractor.

For purposes of comparison, we computed the accuracy and cost for each evaluation run. We measured cost as the number of field labels provided during the labeling phase. We believe this count correlates well with the amount of time it would take a human user to provide the labels requested by active sampling. The CRF sometimes asks the user to label prose text but ListReader never does. To be consistent in measuring cost, we do not count these labelings against the cost for the CRF. This means that the CRF has a slight advantage as it receives training data for negative examples (prose text) without affecting its measured cost. During the test phase, the evaluation system measured the accuracy of the extractors only on portions of text for which the systems did not query the user for labels during training (active sampling).

Since our aim is to develop a system that accurately extracts information at a low cost to the user, our evaluation centers on a standard metric in active learning research that combines both accuracy and cost into a single measurement: Area under the Learning Curve (ALC) [14]. The rationale is that not everyone will agree on a single, fixed level of cost or budget for all information extraction projects. Therefore, the ALC metric gives an average learning accuracy over many possible budgets. We primarily use F_1-measure as our measure of extraction accuracy, which is the harmonic mean of precision and recall, although we also report ALC for precision and recall curves. The curve of interest for an extractor is the set of that extractor's accuracies plotted as a function of their respective costs. The ALC is the percentage of the area between 0% and 100% accuracy that is covered by the extractor's accuracy curve. ALC is equivalent to taking the mean of the accuracy metric at all points along the curve over the cost domain—an integral that is generally computed for discrete values using the Trapezoidal Rule,[11] which is how we compute it. To generate a smooth learning curve for the CRF, we first trained and executed it 7,300 times as explained above, varying the amount of training data supplied to it during its training phase to evenly spread the resulting costs within the chosen cost domain (0 to 90 labeled fields). Next, we applied local polynomial regression to the 7,300 points using "lowess" (locally weighted scatterplot smoother), a function built into the R software environment for statistical computing and graphics. Finally, we computed area under the regression curve, again using the Trapezoidal Rule. We summarize our metrics in Table 4.4.

Table 4.4: Metrics.

Precision = $p = \frac{tp}{tp+fp}$	F-measure = $F_1 = \frac{2pr}{p+r}$
Recall = $r = \frac{tp}{tp+fn}$	ALC = $\int_{min}^{max} f(c)dc$

tp = true positives, *fp* = false positives, *fn* = false negatives
$c =$ Number of user-labeled fields (cost)
$f(c) =$ Precision, recall, or F-measure as a function of cost
min and *max*: smallest and largest number of hand-labeled fields

4.6.4 Results

Figure 4.15 shows a plot of the F-measure labeling/learning curves verses the number of hand-labeled fields for ListReader and the CRF. Visually, the comparative area under the curves (ALC) indicates that ListReader (both one- and two-phase versions) outperforms the CRF uniformly over the number of field labels, and especially for fewer labels during earlier labeling cycles. Statistically, Table 4.5 tells us that these observations are significant ($p < 0.01$, using an unpaired t test). In terms of the ALC of F-measure, the one-phase ListReader outperforms the CRF by 9.1 percentage points, while the two-phase ListReader outperforms the one-phase ListReader by 6.5 percentage points. Table 4.5 also shows that both versions of ListReader perform better than the CRF in terms

[11] See http://en.wikipedia.org/wiki/Trapezoidal_rule

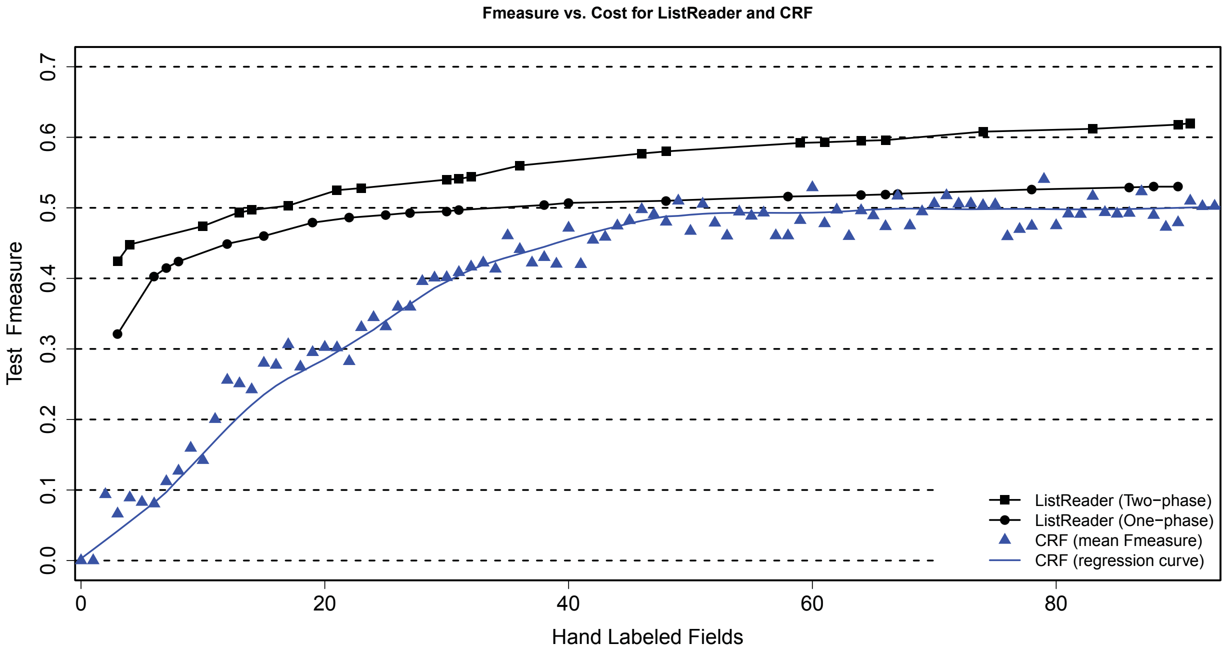


Figure 4.15: Labeling/Learning Curves of ListReader and CRF.

of the components of F-measure, precision and recall, except in the case of recall for one-phase ListReader. The plot of the learning curves for precision in Figure 4.16, shows the highly significant ALC differences for precision ($p < 0.0001$), which is especially good from the very beginning of labeling and which holds when comparing the CRF to either one-phase or two-phase ListReader.

Table 4.5: ALC of Precision, Recall, F-measure (%).

	Prec.	Rec.	F_1
CRF	54.1	34.4	39.5
ListReader (One-phase)	**95.6**	32.7	48.6
ListReader (Two-phase)	94.4	**39.0**	**55.1**

All differences are statistically significant at $p < 0.01$ using an unpaired t test.

In our experimental evaluation, two-phase ListReader produced very few false positives (precision errors), achieving 94.8% precision after the first query cycle and slowly rising to 96.4% by the last query. False negatives (recall errors) were more common. We discuss them, along with future work, below. High precision is a positive result despite the low recall, especially in the context of our aim of reducing the cost of human labor associated with the extraction of information.

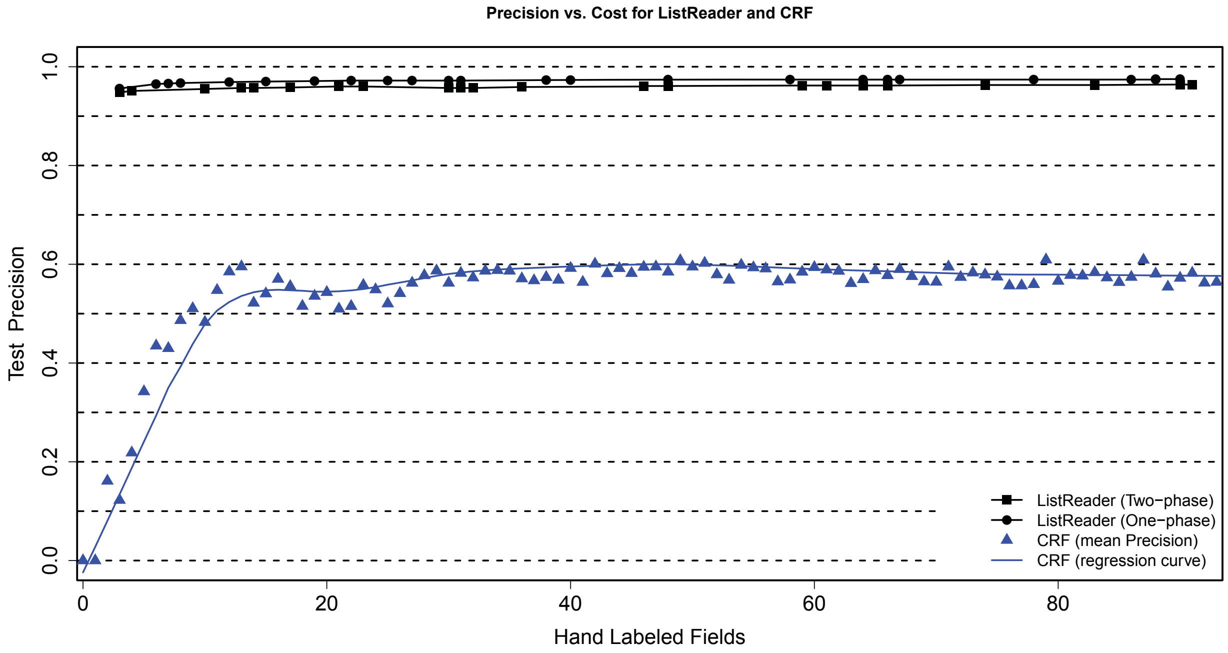


Figure 4.16: Labeling/Learning Curves of ListReader and CRF.

Achieving higher precision means less human post-processing will be needed to correct errors. Achieving higher recall at the expense of lower precision would increase the human time cost, and could even increase it to the point that the automated extraction process ceases to provide any time-saving benefits compared to the user manually extracting all the information. High precision renders ListReader immediately useful in practice for search applications in which sifting through many incorrect results becomes more of a bother than the few good results are worth. For family-history applications, for example, it would be highly bothersome to send email alerts to users reporting finds of information about their ancestors if many of the alerts were false positives. Conversely, highly precise finds would be highly interesting to subscribers of the service.

ListReader generated a regular expression that is 1,473,490 characters long and contains 71,090 capture groups for the *Shaver-Dougherty Genealogy* book. The longest working regex we are aware of is only 157,000 characters long,[12] [13] making our working regular expression nearly 10 times longer. ListReader derived 43,600 of the 71,090 capture groups from second-phase

[12] http://www.terminally-incoherent.com/blog/2007/08/24/biggest-regex-in-the-word/

[13] http://stackoverflow.com/questions/2245282/what-is-the-longest-regular-expression-you-have-seen

record templates. In 35 cycles of active sampling, ListReader collected labels for 218 of those capture groups. All but four of these labeled capture groups were associated with second-phase record templates. This points out the necessity of ListReader's careful strategy of selecting capture groups to label. Using its predicted ROI selection strategy (described in Section 4.5.12), ListReader extracted nearly half of the list-structured information in the pages of the Shaver book by requesting labels for only 0.3% of the capture groups of our generated regex. Of the 218 capture groups, 90 were actually assigned field labels; the rest of the 218 capture groups were field delimiters, page header text, or some other kind of text within a pattern that was not assigned a field label. Many capture groups never need to be labeled by a user for other reasons. About one in thirty do not need a label because they are larger capture groups containing smaller nested capture groups. About the same number again do not need a label because they represent record delimiters.

ListReader's time and space complexity is linear in terms of the size of the input text and the label alphabet (as described in Section 4.5) which also makes it linear in the length of the record patterns. The typical implementation of the training phase of a linear chain CRF is quadratic in both the sizes of the input text and the label set [17], [31]. In our experiments, the CRF occasionally ran out of memory when running on just 41 pages of input. When considering whether ListReader will scale to multiple books, simultaneously, its scalability within a single book is the primary issue. The ListReader process can be easily parallelized and run on a cluster of machines. Each instance can process books as large as *The Ely Ancestry* (which contains well over 800 pages of text) even on a single desktop machine containing only 3.25 GB of RAM. We can therefore split a corpus of many books into individual books and run each instance of ListReader independently of each other. As for wall-clock time, ListReader's unsupervised grammar induction steps (Steps 2 to 11) took 109 seconds, and each active sampling cycle (in Step 12) took 13 seconds, running with JDK 1.7 on a desktop computer with a 2.39 GHz processor and 3.25 GB of RAM. We have not done any work yet to optimize performance other than the high-level design of the described pipeline.

4.7 Error Analysis Leading to Future Work

Analyzing ListReader's precision and recall errors shows us some interesting future research opportunities. Precision is already high, but realizing that the cause of the few errors has to do with incorrectly identifying the beginnings and ends of some records leads to opportunities for future improvements in two ways, which we discuss below: (1) join and label across adjacent record-fragment patterns and (2) split and label patterns spanning multiple adjacent records. Recall is low, leaving considerable room for improvement. In addition to correcting record boundary mistakes, we have three ways to improve recall, which we discuss below: (1) generate an HMM wrapper to go along with the regex wrapper as we did in previous papers [48], (2) address brittleness in the unsupervised grammar induction steps, and (3) automatically propose labels for some unlabeled patterns by aligning fields shared among two or more field groups.

Adjacent Record Fragments The only precision errors appear to happen at record boundaries where a whole record is split into two patterns. This happens because one of the lines of many parent records in the *Shaver-Dougherty Genealogy* book share a conflated text profile that also satisfies the record selection criteria. For example, in the following text, ListReader correctly labels the second half of each record without grouping that pattern with the first half of the records.

```
16-1-1-3-15-5-2^ Ray, Donald Allen-b. 23 Nov 1939-mar.
Perrow, Susan-d. nr-ch. 5:

16-1-1-3-15-7-1 Shafer, Philip Elvin-b. 14 Jun 1949-mar
Myers, Fay-d. nr-ch. 2:
```

That causes a precision error, a false positive record boundary between the two lines. This also produces false negatives when ListReader does not go on to recognizing the first half of these records. Currently, the structure of the grammar learned before active sampling begins is held fixed during active sampling—only the labels of capture groups change. ListReader could learn to join two patterns based either on a smarter unsupervised pattern selection strategy during the record selection steps or allowing the user to manually group adjacent strings in the active sampling step.

Related Adjacent Records Like record fragments, full records may also be adjacent. For example, in Figure 4.1 the parent records are adjacent to the child records. In our current work we did not expect ListReader to be able to relate elements across record boundaries, but in future work we would like to do so and thus in Figure 4.1 be able to relate parents in family header records with children in child records. As for adjacent record fragments, ListReader may be able to join or label across full record patterns.

HMM Wrapper We plan to construct an HMM in the place of, or along with, the regex described in this paper. Both the regex and HMM can be generated from the patterns we discover in the suffix tree within the unsupervised pipeline. The proposed HMM will be similar to the HMM induced in [48] in some ways, e.g., in explicitly modeling common variations in record structure and in the HMM's flexibility in accepting some OCR errors based on the parameter smoothing of the emission model. The proposed HMM, however, will be different from the HMM of [48] in starting out with no user-specified labels, in its parameters being trained from a much larger set of examples obtained in the unsupervised grammar-induction steps, and therefore in being more robust to randomness in OCR text. These aspects of the HMM should improve recall considerably. We expect that precision will not decrease very much; but if precision becomes a problem, we can continue to rely on the regex wrapper wherever it applies.

Brittleness Brittleness in some of the unsupervised grammar-induction steps prevents ListReader from detecting patterns that occur infrequently. This is because ListReader relies on predetermined values of parameters (cut-off values). For example, in the step that selects field group delimiters, not only can we not assume that a single cut-off value will be optimal for all books, we cannot assume that a single cut-off value will be optimal for all candidate delimiters within a single book. ListReader discarded a "was born" delimiter candidate that appeared in only one record cluster. On the other hand, the “un” candidate[14] appears in three clusters and is therefore closer to being accepted as a field group delimiter than “was born”. We believe that a more statistically well-founded approach, e.g., a collocation or hypothesis testing approach, will be better able to identify, not only true field group delimiters, but also the patterns in other steps of the pipeline while remaining completely unsupervised. This could allow for automatic parameter adaptation for conflation rule selection, field group delimiter extraction, record type selection, etc. Simply adding more conflation rules could also decrease brittleness in pattern-finding.

Unlabeled Patterns ListReader does not receive any labels for certain patterns before reaching a predetermined label budget (e.g., 90 field labels). Because of this, many capture groups remain unlabeled. ListReader currently will apply no labels to the text matching those capture groups. We can overcome this limitation by (1) continuing to run additional active sampling cycles or (2) allowing ListReader to propose labels for unlabeled field patterns based on their similarity to known, labeled field patterns. For example, if the user has already labeled a birth date containing a Day, a $Month$, and a $Year$ during the course of active sampling, ListReader could propose the $Year$ label for a date containing only a year instead of leaving this pattern unlabeled. Or, for another example, after labeling one sequence of names, one list of page numbers in an index, or one sequence of ancestor birth-order numbers, label other similar repeating field groups based on the labeling of the first. Knowing how to do this in general for any kind of field while maintaining high precision will take additional research and will likely require ListReader to make more use of the labels it receives during active sampling (e.g., true active learning).

4.8 Conclusions

We see a tendency in research to focus on improving accuracy while ignoring the hidden increases in costs associated with those improvements. Well-developed statistical models such as the CRF can perform well on a number of tasks, but that performance comes with additional costs in terms of

[14] “un” appears in phrases like “un-named son” and “un-identified child”.

knowledge and feature engineering, manual labeling of training data, and other domain-, genre-, and task-specific refinements—not to mention the increasingly specialized knowledge of mathematics and data science needed to re-implement, or even to understand how to apply, the new approach.

Our ListReader project addresses these concerns. Through our ListReader approach to information extraction, we have demonstrated a simple, scalable, and effective way of simultaneously improving the accuracy and decreasing the cost of extracting information from OCRed lists. The work required of a user to perform information extraction is as simple as building an HTML form and filling it several times with the corresponding text as requested by the system. ListReader effectively combines unsupervised grammar induction with active sampling to identify, extract, and structure data in lists of noisy OCRed text. ListReader performs well in terms of accuracy, user labeling cost, time and space complexity, and required knowledge engineering—outperforming the CRF in each of these performance measures. Its precision is high enough for immediate practical use in applications where query results should be precise (but not necessarily complete). Furthermore, and of interest for future research, its precision is high enough that we can also use the output of ListReader as training data for supervised extractors—training data that now comes at no additional cost for the human labeler.

Chapter 5

Unsupervised Training of HMM Structure and Parameters for OCRed List Recognition and Ontology Population

Abstract

Machine-learning-based approaches to information extraction and ontology population often require a large number of manually selected and annotated examples in order to work. In this paper, we evaluate ListReader, which provides a way to train the structure and parameters of a hidden Markov model (HMM) using text selected and labeled completely automatically. This HMM is capable of recognizing lists of records in OCRed and other text documents and clustering related fields across record templates. The training method we employ is based on a novel unsupervised active grammar-induction framework that, after producing an HMM wrapper, uses an efficient active sampling process to complete the mapping from the HMM wrapper to ontology by requesting annotations from a user for automatically-selected examples. We measure performance of the final HMM in terms of F-measure of extracted information and manual annotation cost and show that ListReader learns faster and better than a state-of-the-art baseline (CRF) and an alternate version of ListReader that induces a regular expression wrapper.

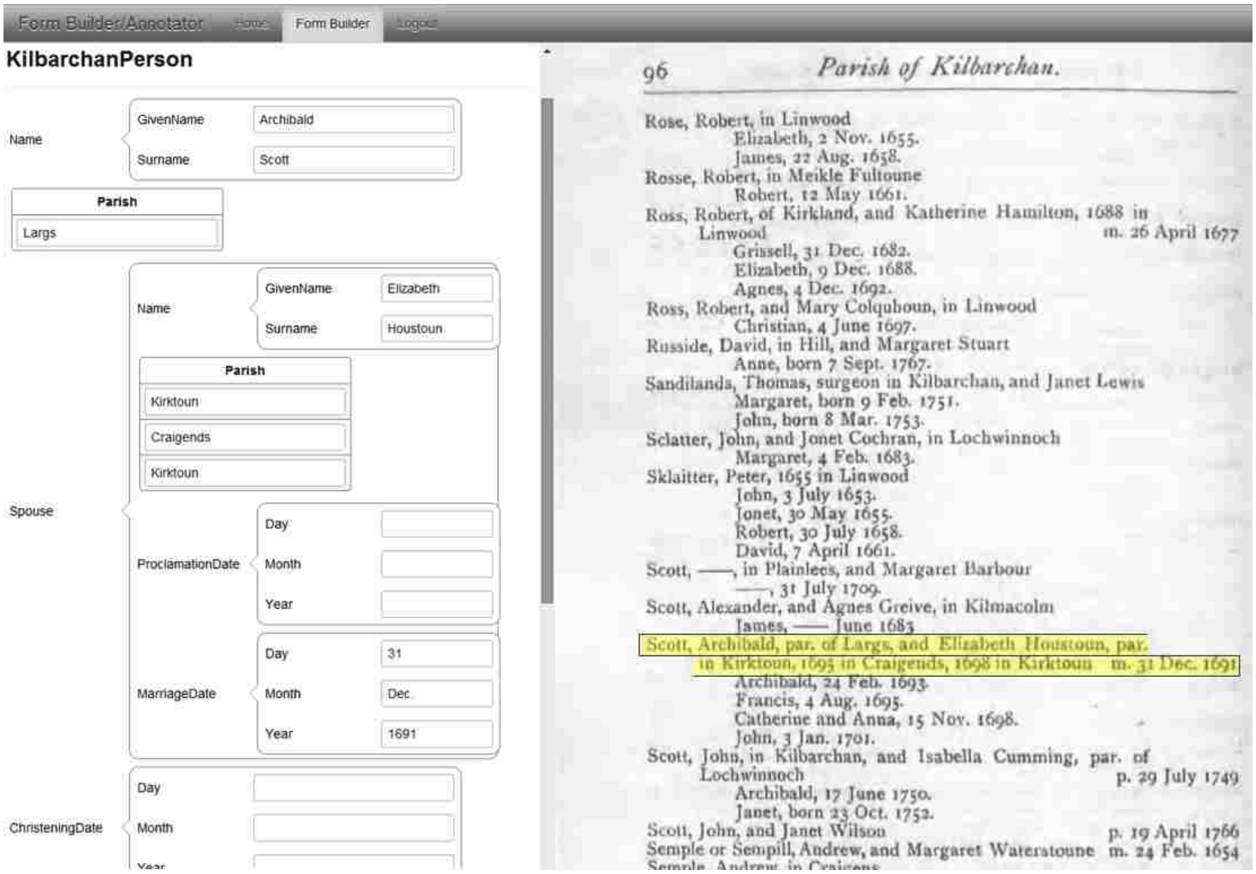
5.1 Introduction

Information extraction and ontology population are areas of research concerned with building models or processes to discover information in implicitly-structured sources like text and to make the structure of that information explicit, machine-readable, and more readily usable by computing machines. Wrapper induction [41] and other machine-learning-based approaches are commonly employed to efficiently produce an extraction model or wrapper. Supervised machine-learning-based approaches are common (e.g. [35], [44]) and can perform well in terms of accuracy, but often require a large amount of experimentation and knowledge engineering to produce an effective set of feature extractors and a large number of manually selected and annotated examples to learn well.

We propose *ListReader*, an unsupervised active wrapper induction process for learning Hidden Markov Models (HMMs) without technical expert input that are customized to the structure of each text document (e.g., a book) and capable of populating one or more richly-structured ontologies. ListReader requires no hand-labeled training data to construct an HMM. It does, however, require a small number of hand-labeled examples and a minimal amount of knowledge engineering to finalize the mapping from HMM-labeled text to a populated ontology. In the end, ListReader induces a wrapper that is more accurate than a standard supervised machine learning approach but requires fewer hand-labeled examples and less knowledge engineering. Moreover, it minimizes the ways in which the hand-labeled examples affect the final model. In particular, since hand-labeled data only affects the external mapping from HMM states to semantic labels, we can more easily repurpose a previously-induced wrapper for a new target ontology.

To start ListReader processing, a user selects a text document containing one or more lists of records, e.g., an OCRed collection of page images from a scanned book selected for an information application. For example, the user could select the *Kilbarchan Parish Register* [26] for a family history application. Part of one page of this book appears in the right side of Figure 5.1. The user constructs a data entry form for the desired information in the left side of the user interface, e.g., the form in Figure 5.1 before being filled in.[1] ListReader translates the form into an ontology

[1] The construction of a form is the full extent of ListReader-required manual knowledge engineering.



KilbarchanPerson

Name		GivenName	Archibald
		Surname	Scott
Parish			Largs
Spouse	Name	GivenName	Elizabeth
		Surname	Houstoun
	Parish		Kirktoun Craigends Kirktoun
	ProclamationDate	Day	
		Month	
		Year	
	MarriageDate	Day	31
		Month	Dec
		Year	1691
ChristeningDate		Day	
		Month	
		Year	

96 *Parish of Kilbarchan.*

Rose, Robert, in Linwood
Elizabeth, 2 Nov. 1655.
James, 22 Aug. 1658.
Rosse, Robert, in Meikle Fultoune
Robert, 12 May 1661.
Ross, Robert, of Kirkland, and Katherine Hamilton, 1688 in
Linwood m. 26 April 1677
Grissell, 31 Dec. 1682.
Elizabeth, 9 Dec. 1688.
Agnes, 4 Dec. 1692.
Ross, Robert, and Mary Colquhoun, in Linwood
Christian, 4 June 1697.
Russide, David, in Hill, and Margaret Stuart
Anne, born 7 Sept. 1767.
Sandilands, Thomas, surgeon in Kilbarchan, and Janet Lewis
Margaret, born 9 Feb. 1751.
John, born 8 Mar. 1753.
Sclatter, John, and Jonet Cochran, in Lochwinnoch
Margaret, 4 Feb. 1683.
Sklaitter, Peter, 1655 in Linwood
John, 3 July 1653.
Jonet, 30 May 1655.
Robert, 30 July 1658.
David, 7 April 1661.
Scott, ——, in Plainlees, and Margaret Barbour
——, 31 July 1709.
Scott, Alexander, and Agnes Greive, in Kilmacolm
James, —— June 1683
Scott, Archibald, par. of Largs, and Elizabeth Houstoun, par.
in Kirktoun, 1695 in Craigends, 1698 in Kirktoun m. 31 Dec. 1691
Archibald, 24 Feb. 1693.
Francis, 4 Aug. 1695.
Catherine and Anna, 15 Nov. 1698.
John, 3 Jan. 1701.
Scott, John, in Kilbarchan, and Isabella Cumming, par. of
Lochwinnoch p. 29 July 1749
Archibald, 17 June 1750.
Janet, born 23 Oct. 1752.
Scott, John, and Janet Wilson p. 19 April 1766
Semple or Sempill, Andrew, and Margaret Wateratoune m. 24 Feb. 1654
Semple, Andrew, in Craigens

Figure 5.1: *Kilbarchan Parish Register* Page and KilbarchanPerson Filled-in Form.

schema, e.g., the target ontology in Figure 5.2. Without anything more than the given text document (e.g., the entire collection of page images of the Kilbarchan book in our example), ListReader applies an unsupervised process to automatically discover and align records, induces a simple phrase structure grammar, and trains the structure and parameters of an HMM. After ListReader sets the HMM's structure and parameters, it actively requests labels for selected strings of text from the user. ListReader highlights the strings it selects for the user to label, as Figure 5.1 shows. The user provides labels by filling in the data entry form.[2] Figure 5.1 shows the filled-in form for the highlighted text. ListReader uses the structure of the form to generate specialized labels for the field strings in the text document that specify the mapping of the strings to ontology predicates. Figure 5.3 shows the labels for the highlighted record. After labeling, the structure and parameters

[2] Filling in a form with ListReader-selected text is the full extent of required hand labeling.

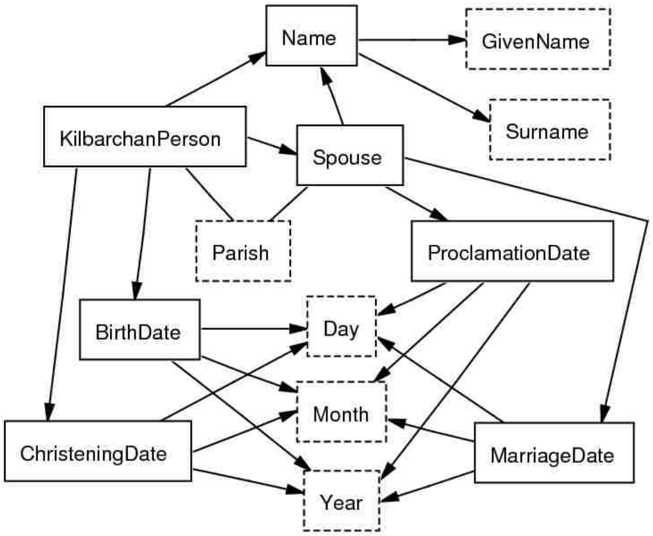


Figure 5.2: KilbarchanPerson Ontology.

```
<KilbarchanPerson.Name.Surname>Scott</KilbarchanPerson.Name.Surname>,
<KilbarchanPerson.Name.GivenName>Alexander</KilbarchanPerson.Name.GivenName>
, par. of <KilbarchanPerson.Parish[1]>Largs</KilbarchanPerson.Parish>[1], and
<KilbarchanPerson.Spouse.Name.GivenName>Elizabeth</KilbarchanPerson.Spouse.Name.GivenName>
<KilbarchanPerson.Spouse.Name.Surname>Houstoun</KilbarchanPerson.Spouse.Name.Surname>
, par. in <KilbarchanPerson.Spouse.Parish[1]>Kirktoun</KilbarchanPerson.Spouse.Parish>[1]>
, 1695 in <KilbarchanPerson.Spouse.Parish[2]>Craigends</KilbarchanPerson.Spouse.Parish>[2]
, 1698 in <KilbarchanPerson.Spouse.Parish[3]>Kirktoun</KilbarchanPerson.Spouse.Parish>[3]
m. <KilbarchanPerson.Spouse.MarriageDate.Day>13</KilbarchanPerson.Spouse.MarriageDate.Day>
<KilbarchanPerson.Spouse.MarriageDate.Month>Dec.</KilbarchanPerson.Spouse.MarriageDate.Month>
<KilbarchanPerson.Spouse.MarriageDate.Year>1691</KilbarchanPerson.Spouse.MarriageDate.Year>
```

Figure 5.3: Labeled Record of the Highlighted Text in Figure 5.1.

of the HMM are unchanged but some of the states will have been assigned labels by the user. ListReader executes the final HMM using the Viterbi algorithm and maps labeled text to predicates, thus completing the mapping from text to ontology.

Our approach to wrapper induction is a combination of unsupervised learning and active learning. ListReader is *unsupervised* in that it induces an HMM without labeled training data and does not alter the structure or parameters of this HMM after it starts making *active* requests of the user for labels which it receives and assigns to existing HMM states. Because of how the HMM is induced, one label from the user may apply to more than one HMM state, which greatly reduces the amount of required labeling. Furthermore, ListReader follows the spirit of active learning [53] in that it uses this structural model to request labels for those corresponding parts of the known and unlabeled text that will have the greatest impact on the final wrapper's mapping from text to ontology, meaning the greatest increase in recall for the lowest number of hand-labeled fields.

Our ListReader research makes the following contributions. First, we provide an algorithm to train both model structure and parameters of an HMM for list recognition without hand-labeled examples. This algorithm is linear in time and space with respect to the input text length, the discovered pattern length, and output label alphabet. Second, we provide an efficient active sampling process to complete the HMM as a data-extraction wrapper that can map the data in lists to an expressive ontology schema. Active sampling is an active-learning-like process that requests labels of selected examples from the user without modifying the internal wrapper structure. The final wrapper outperforms two alternatives in terms of a metric that combines precision, recall, and annotation cost. Our global approach to pattern discovery in the first contribution is complementary to the active sampling process in the second in that ListReader first discovers the most frequent patterns which then produce the greatest return on investment of the user's time in annotating.

We give the details of these contributions as follows. In Sections 5.2, 5.3, and 5.4 we describe the HMM wrapper induction process, illustrating the steps with a running example of the execution of ListReader on the 140-page *Kilbarchan Parish Record* [26]. We explain in Section 5.2 how ListReader discovers record-like patterns in text in linear time and space and in Section 5.3 how ListReader derives the structure and parameters of an HMM from the discovered patterns—both without human supervision. In Section 5.4 we tell how ListReader creates the mapping from HMM states to an ontology using active sampling. In Section 5.5 we provide an evaluation of the performance of ListReader in terms of the precision, recall, and F-measure of the automatically extracted information, each as a function of manual field labeling cost, and compare the learning rates to a state-of-the-art statistical sequence labeler (CRF) and to a previous version of ListReader that induces regular-expression-based wrappers. In Section 5.6 we discuss performance issues and opportunities for future work, and in Section 5.7 we compare our proposed solution to related work on information extraction from lists and on unsupervised wrapper induction. Finally, we make concluding remarks in Section 5.8.

5.2 Unsupervised Pattern Discovery

In its unsupervised process of pattern discovery, ListReader finds record-like patterns in the input text, produces a representation of the hierarchical field structure of these strings, and associates the major components (delimited field groups) across different types of records. In the next major step, detailed in Section 5.3, ListReader flattens this hierarchical structure into a state machine and sets the parameters of the HMM using statistics in the collection of parsed record patterns. As we explain in Subsection 5.2.1, ListReader begins to discover patterns by conflating the input text—substituting abstract word and phrase structure for strings of characters. ListReader then efficiently identifies record-like patterns in the abstract text (Subsection 5.2.2). Subsequently, ListReader further parses and aligns field groups within and between record patterns (Subsection 5.2.3). Finally, ListReader establishes the set of record and field group templates from which it constructs the HMM (Subsection 5.2.4).

5.2.1 Text Conflation

ListReader converts input text into an abstract representation using a small pipeline of conflation rules. They perform tokenization and chunking of the text which in turn improve tolerance of many common OCR errors and the natural variations among fields of the same type. Currently, we have established the following conflation rules, given in their order of application.

1. *Split Word*: Merges two alphabetic word tokens that are separated by a hyphen and a newline into a single word symbol.
2. *Horizontal Punctuation*: Conflates thin, horizontally-oriented punctuation characters: underscore, hyphen, en dash, em dash, and other Unicode variations.
3. *Numeral*: Replaces each digit in a numeral with a digit designator (“`Dg`”).
4. *Word*: Replaces contiguous letters with a generic word designator that only preserves the relative order of upper-case (“`Up`”) and lower-case (“`Lo`”) characters. ListReader optionally preserves the full spelling of lower-case words.

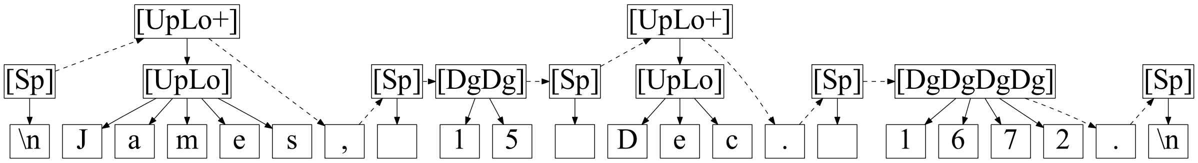


Figure 5.4: Initial Parse of "\nJames, 15 Dec. 1672.\n".

5. *Space*: Conflates normal space characters (“ ”) with newlines (“\n”) using a common symbol (“[Sp]”).

6. *Incorrect Space*: Removes spaces that occur on the “wrong side” of certain punctuation characters because of an OCR or typesetting error, such as immediately before a period.

7. *Capitalized Word Repetition*: Replaces each contiguous sequence of space-delimited capitalized words of any length with a single, generic symbol (“[UpLo+]”).

8. *Numeral Repetition*: Replaces each contiguous sequence of comma- or hyphen-separated numerals of any length greater than one with a generic symbol (e.g. “[Dg+-]”) that only preserves the identity of the punctuation delimiter. The same delimiter must be found between every pair of numerals in a sequence. Extra spaces around the punctuation delimiter are also allowed.

The cumulative application of the conflation rules produces a sequence of small parse trees. Figure 5.4 shows the parse trees for the text “\nJames, 15 Dec. 1672.\n”. The dashed line joins the root phrase symbols giving a new sequence of symbols in which ListReader looks for patterns. Not all conflation rules need be used for every book. Generally, all of them should be used that do not erase distinctions between records that should not be aligned, such as is the case when conflating lower-case words in records where different field group delimiters like “born on” and “died on” are aligned. Preventing the conflation of lower-case words is appropriate for books such as the *Kilbarchan Parish Register* [26] which contains very little prose and whose list records are consistently structured.

5.2.2 Record Pattern Search

To find record-like patterns in the simplified text, ListReader first builds a suffix tree data structure from the conflated text. It then searches for repeated patterns that satisfy our record-selection constraints: *records must begin and end with a valid record delimiter, must occur at least twice, and must contain at least one numeral or capitalized word.*

ListReader relies on the suffix tree to efficiently find patterns in text. A *suffix tree* is a compact data structure representing all suffixes of a text string by paths from its root to its leaf nodes. (In our running example, the text is the single long string of conflated symbols from the first conflation symbol for the first token on the first page of the book to the last conflation symbol for the last token on the last page.) Each edge in the suffix tree is labeled by the substring of symbols it represents. In addition, the information maintained for each edge includes the offsets for each occurrence of the edge’s substring and a count of these occurrences. ListReader constructs a suffix tree in linear time and space using Ukkonen’s algorithm [58]. It also finds and collects record-like patterns within the suffix tree in linear time and space by iterating over the edges of the suffix tree and checking for strings terminating in each edge that adhere to the properties specified for a record.

Figure 5.5 shows several patterns ListReader finds from the conflated text of the *Kilbarchan Parish Register*. With each pattern we also give in Figure 5.5 the number of unconflated strings that belong to the pattern and show a few of them—all of them for the last pattern. Observe that all of the strings of original text satisfy the properties required of records—each begins and ends with “`\n`” (a specified record delimiter) and includes a number or a capitalized word or both. The patterns are thus potential record patterns, and each pattern along with its candidate records forms a *record cluster*. Its pattern is called a *record template*.

5.2.3 Field Group Discovery

ListReader next discovers parts of records (field groups) that recur among different record clusters. These correspondences will be represented later in the HMM and used to reduce the number of necessary hand-labeled fields. The intuition is that since a field like a birth year or marriage year

```
[[Sp][UpLo+],[Sp][DgDg][Sp][UpLo+].[Sp][DgDgDgDg].[Sp]]
Record instance count: 1296
    \nJames, 15 Dec. 1672.\n
    \nRobert, 15 Oct. 1676.\n
    ...

[[Sp][UpLo+],[Sp][DgDg][Sp][UpLo+][Sp][DgDgDgDg].[Sp]]
Record instance count: 710
    \nJoan, 25 April 1651.\n
    \nJohn, 30 May 1652.\n
    ...

[[Sp][UpLo+],[Sp][born][Sp][DgDg][Sp][UpLo+].[Sp][DgDgDgDg].[Sp]]
Record instance count: 441
    \nWilliam, born 10 Dec. 1755.\n
    \nJames, born 24 Oct. 1758.\n
    ...

[[Sp][UpLo+],[Sp][born][Sp][DgDg][Sp][UpLo+][Sp][DgDgDgDg].[Sp]]
Record instance count: 265
    \nWilliam, born 23 June 1747.\n
    \nJames, born 19 June 1749.\n
    ...

[[Sp][UpLo+],[Sp][UpLo+],[Sp][and][Sp][UpLo+][Sp][m].[Sp][DgDg][Sp][UpLo+].[Sp]
[DgDgDgDg][Sp]]
Record instance count: 61
    \nAiken, David, and Janet Stevenson m. 29 Sept. 1691\n
    \nAitkine, Thomas, and Geills Ore m. 21 Dec. 1661\n
    ...

[[Sp][UpLo+],[Sp][UpLo+],[Sp][par].[Sp][of][Sp][UpLo+],[Sp][and][Sp][UpLo+][Sp]
[m].[Sp][Dg][Sp][UpLo+].[Sp][DgDgDgDg][Sp]]
Record instance count: 6
    \nBarbor, Ninian, par. of Lochwinnoch, and Marion Reid m. 3 Mar. 1681\n
    \nBarbor, William, par. of Paisley, and Elizabeth Gibson m. 9 Dec. 1680\n
    \nCrafurd, Thomas, par. of Beith, and Catherine Wilsoune\nm. 6 Sept. 1660\n
    \nErskine, John, par. of Lochwinnoch, and Jonet Reid m. 8 Dec. 1658\n
    \nInglis, John, par. of Glasgow, and Annas Shaw m. 6 Jan. 1660\n
    \nLyle, John, par. of Kilmacome, and Jessie Cochran m. 8 Feb. 1677\n
```

Figure 5.5: A Selection of Record Clusters from the *Kilbarchan Parish Register*.

follows a specific field group delimiter like “born” or “m.”, it can be identified and labeled the same even when found in different record clusters. The dates following ”born” in the second and third record cluster in Figure 5.5 are all birth dates, and the dates following “m.” in the last two record clusters are all marriage dates.

From the set of aligned record clusters discovered, ListReader identifies *field group delimiters*, defined as follows:

1. *sequences of lower-case words separated by whitespace or punctuation that occur in a fixed position within a few different record clusters along with their adjacent whitespace and punctuation characters.* Requiring “few” to be more than four record clusters typically eliminates valid delimiters from consideration, and requiring less than two record clusters provides insufficient evidence for delimiter patterns. From the clusters in Figure 5.5, ListReader can identify the following field group delimiters of this first type: “, born ”, “, and ”, and “ m. ”. With additional supporting evidence from other clusters in the book, ListReader would also identify “, par. of ” in the last record cluster in Figure 5.5 as a field group delimiter.
2. *record delimiters along with whitespace and punctuation that follow or precede them.* In Figure 5.5, all the initial record delimiters (i.e. “\n”) and both types of final record delimiters (i.e. “.\n” and “\n”) are field group delimiters of this second type.

ListReader constructs *field group templates* from the text appearing between field group delimiters and associates a field group template with the delimiter on its left. Field group templates consist of a field group delimiter, whose text *is not* conflated in the template, followed by one or more variations of the field group itself, whose text *is* conflated. For example, the field group template for the delimiter “, born ” in the third record cluster in Figure 5.5 is “, born [DgDg][Sp][UpLo+].[Sp][DgDgDgDg]” and one of its variations in the fourth record cluster in Figure 5.5 is “, born [DgDg][Sp][UpLo+][Sp][DgDgDgDg]”. In the third record cluster the months are abbreviations followed by a period, and in the fourth the months are spelled out and have no period. Additional variations in the *Kilbarchan Parish Record* include dates with single-digit days, dates with missing days (only months and years), and dates with day ranges, presumably when the birth day is only approximately known. The " m. " delimiter from the last two record clusters in Figure 5.5 has only one variation in the figure, but in the full *Kilbarchan Parish Record*, the dates have several variations. The initial delimiter, "\n" has the following variations in Figure 5.5:

```
[UpLo+],[Sp][DgDg][Sp][UpLo+].[Sp][DgDgDgDg]
[UpLo+],[Sp][DgDg][Sp][UpLo+][Sp][DgDgDgDg]
[UpLo+]
[UpLo+],[Sp][UpLo+]
```

and many more in the full book. A field group template for a final record delimiter is just the delimiter itself as no field follows it.

The delimiters themselves also allow for slight variations. Because of OCR or typesetting errors, "m. " may sometimes appear as "m, " or "m: ", for example. In the *Kilbarchan Parish Record"*, when a name in records like "\nJames, 15 Dec. 1672.\n" is unknown, the typesetters let a long dash represent the unknown name, e.g. "\n———, 15 Dec. 1672.\n". In this case the OCR treats the long dash as line art and ignores it, but does pick up the comma making the initial record delimiter have "\n, " as a variation. ListReader recognizes delimiter variations by considering any delimiter text that contains the same sequence of lower-case words (or newlines in the special case of record delimiters) as the same delimiter despite any variations in punctuation and spaces.

5.2.4 Final Record and Field Group Template Selection

At this point ListReader almost has what it needs for HMM creation. With some additional adjustments, ListReader will have identified record and field group templates from which it can directly construct an HMM that will extract the fields in the records. The adjustments include discarding record patterns that do not resolve into a clean sequence of field group templates and grouping record clusters that satisfy the same sequence of field group templates and then splitting some of the individual field group templates into alternate template groups depending on whether there is enough variation to warrant a split.

Figure 5.6 shows the new record clusters that include the record clusters in Figure 5.5 after grouping clusters with the minor variations. The first group in Figure 5.6 includes the records in the first two clusters in Figure 5.5, and many more—3341 altogether, which includes not only the 1296 in the first cluster and 710 in the second, but also all others that have the pattern "`\n<name>, <date>.\n`". The second cluster in Figure 5.6 groups the third and fourth clusters in Figure 5.5 as well as several other clusters that all have the pattern "`\n<name>, born <date>.\n`". The third cluster in Figure 5.6 groups the fifth cluster in Figure 5.5 with others like it, and the last cluster in Figure 5.6 shows the complete grouping of records for the last cluster in Figure 5.5. Note how the added records vary from the six in Figure 5.5: double-digit days, months that are not abbreviated, names with punctuation ("`M'Gregor`"), and names with a missing surname (the last two). Being complete and small enough will allow this last grouping to serve as an example for the field-group-template-splitting adjustment ListReader makes.

First, however, we explain how ListReader groups record clusters and ensures that each pattern consists of a clean sequence of field group templates. When used to parse text, we say that a field group template produces a *field group segment* as a new type of parse tree node. These nodes can be seen in the parse trees in Figure 5.7 for the first string of characters constituting the first record in the third cluster group in Figure 5.6. Except for the special "End-Segment" node, each "Segment" node includes a "Delim" node followed by a "FieldGroup" node as Figure 5.7 shows.

As a next step, ListReader again produces a suffix tree, working with a new input sequence composed of both field group segments wherever a field group template matches text and the original conflated text elsewhere. In Figure 5.7 the text matches field group templates, and thus the sequence of symbols from which ListReader constructs this second suffix tree is a sequence of "`[...-Segment]`" symbols—"`[\n-Segment][and-Segment][m-Segment][\n-End-Segment]`" for the text in Figure 5.7. On the other hand, when the text of a potential record does not fully parse into "`[...-Segment]`" nodes as is the case for the three record

```
[[\n-Segment][\n-End-Segment]]
Record instance count: 3341
    \nJames, 15 Dec. 1672.\n
    \nRobert, 15 Oct. 1676.\n
    ...
    \nJoan, 25 April 1651.\n
    \nJohn, 30 May 1652.\n
    ...

[[\n-Segment][born-Segment][\n-End-Segment]]
Record instance count: 1078
    \nWilliam, born 10 Dec. 1755.\n
    \nJames, born 24 Oct. 1758.\n
    ...
    \nWilliam, born 23 June 1747.\n
    \nJames, born 19 June 1749.\n
    ...

[[\n-Segment][and-Segment][m-Segment][\n-End-Segment]]
Record instance count: 132
    \nAiken, David, and Janet Stevenson m. 29 Sept. 1691\n
    \nAitkine, Thomas, and Geills Ore m. 21 Dec. 1661\n
    ...

[[\n-Segment][par-of-Segment][and-Segment][m-Segment][\n-End-Segment]]
Record instance count: 23
    \nBarbor, Ninian, par. of Lochwinnoch, and Marion Reid m. 3 Mar. 1681\n
    \nBarbor, William, par. of Paisley, and Elizabeth Gibson m. 9 Dec. 1680\n
    \nBarr, John, par. of Killelan, and Issobell Cunynghame m. 12 July 1661\n
    \nBlair, Hugh, par. of Kilmacome, and Margaret Roger m. 22 Aug. 1677\n
    \nCarruth, John, par. of Kilmacolm, and Jean Houstoun m. 25 Nov. 1656\n
    \nCochran, William, par. of Lochwinnoch, and Jonet King m. 13 May 1675\n
    \nCraig, John, par. of Beith, and Marione Speir m. 18 Dec. 1672\n
    \nErskine, John, par. of Lochwinnoch, and Jonet Reid m. 8 Dec. 1658\n
    \nInglis, John, par. of Glasgow, and Annas Shaw m. 6 Jan. 1660\n
    \nKelloch, Mungo, par. of Kilmalcome, and Jonet Andrews m. 24 Feb. 1657\n
    \nLang, John, par. of Kilmacome, and Mary Love m. 15 Feb. 1677\n
    \nLochhead, John, par. of Nilston, and Helen Rodger m. 5 May 1681.\n
    \nLyle, John, par. of Kilmacome, and Jessie Cochran m. 8 Feb. 1677\n
    \nM’Gregor, John, par. of Kilmacome, and Isobel Flemyng m. 22 Dec. 1673\n
    \nMudie, John, par. of Kilmacome, and Jonet Lyle m. 25 April 1678\n
    \nOre, John, par. of Paisley, and Elizabeth How m. 29 Oct. 1650\n
    \nPaterson, John, par. of Paisley, and Janet Caldwell m. 14 June 1652\n
    \nShaw, John, par. of Erskine, and Jean Mudie m. 28 Jan. 1651\n
    \nSmith, David, par. of Lochwinnoch, and Agnes Hair m. 30 Nov. 1660\n
    \nWallace, John, par. of Paisley, and Agnes Lennox m. 23 Dec. 1680\n
    \nWilson, Patrick, par. of Paisley, and Jonet Thomson m. 21 Dec. 1676\n
    \n, William, par. of Kilpatrick, and Issobel Dalzel m. 26 Oct. 1655\n
    \n, Robert, par. of Lochwinnoch, and Issobell King m. 17 July 1673\n
```

Figure 5.6: Record Clusters Grouped by Clean Field Group Template Sequences.

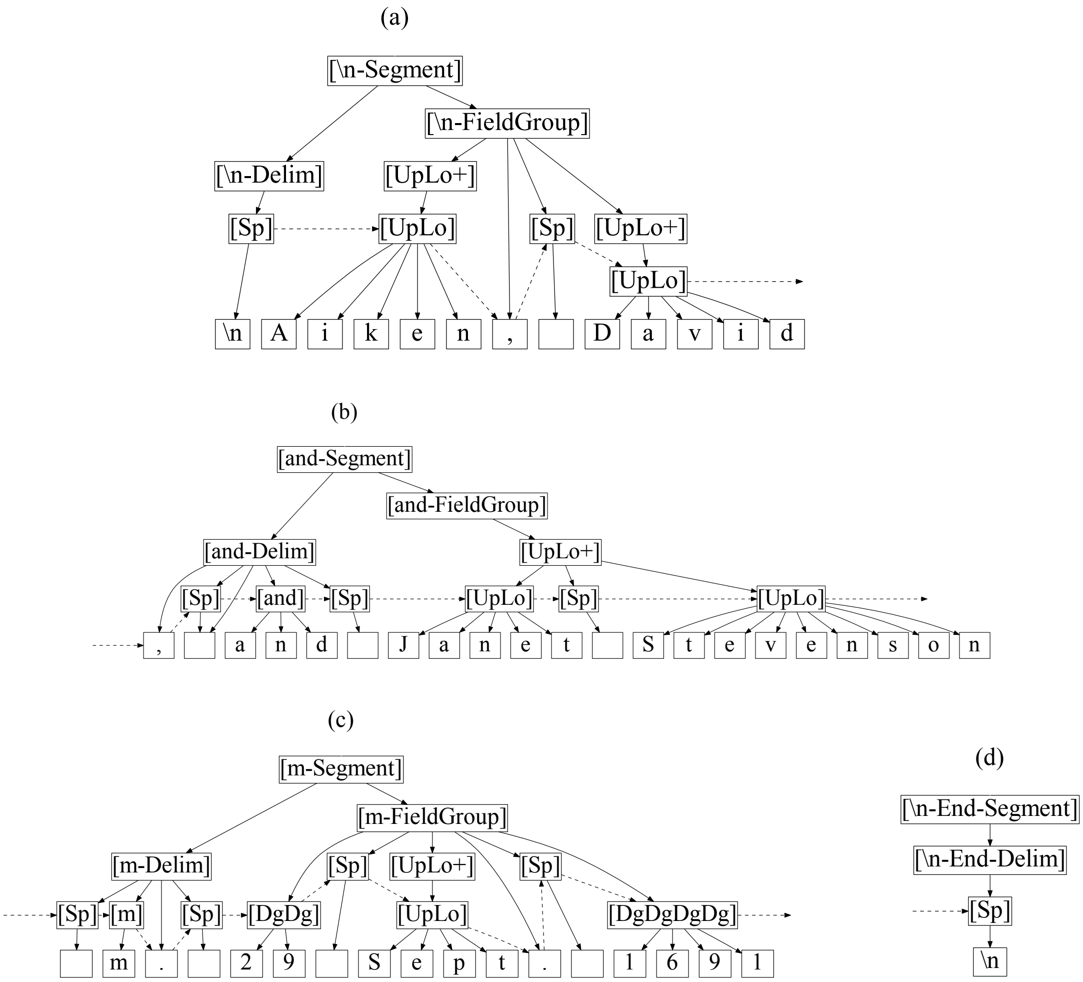


Figure 5.7: Parse Trees of "\nAiken, David, and Janet Stevenson m. 29 Sept. 1691\n" for each Segment: (a) "[\n-Segment]", (b) "[and-Segment]", (c) "[m-Segment]", and (d) "[\n-End-Segment]".

clusters in Figure 5.8, the sequence of symbols is the original conflated text (e.g. the sequence of conflation symbols at the top of each record cluster in Figure 5.8). The first record cluster in Figure 5.8 fails to parse into a sequence of "`[...-Segment]`" symbols because of the inserted birth times, the second because of the deleted day in the date, and the third because of the OCR errors, substituting letter characters for digits.

```
[\n,[Sp][born][Sp][DgDg][Sp][UpLo+][Sp][Dg+,][Sp][a].[m]\n]
Record instance count: 2
    \nBarbara, born 10 July 1763, 3 a.m.\n
    \nMary, born 28 July 1750, 2 a.m.\n

[\n.[Sp][DgDgDgDg]\n]
Record instance count: 2
    \nJames, Dec. 1656.\n
    \nRobert, Nov. 1689.\n

[\n,[Sp][Up][Sp][UpLo+].[Sp][DgDgDgDg]\n]
Record instance count: 3
    \nWilliam, JO Sept. 1758.\n
    \nElizabeth, I Nov. 1700.\n
    \nJohn, II Mar. 1705.\n
```

Figure 5.8: Record Clusters without Field Group Segment Matches.

ListReader searches for record patterns in this second suffix tree as it does in the first suffix tree with one additional constraint: each record template must be composed entirely of field group segments. Thus, ListReader clusters all the text strings of record candidates that have the same sequence of "`[...-Segment]`" templates despite any variations in the underlying text. The record clusters for the text string in Figure 5.7 and hundreds of other similar text strings in the *Kilbarchan Parish Record* are all grouped together in the same cluster. This clustering produces a higher level of text abstraction and a better identification of basic patterns from which to build HMM recognizers (e.g. the higher-level record patterns in Figure 5.6 rather than the lower-level record patterns in Figure 5.5). Furthermore, rejecting some clusters like those in Figure 5.8 helps ListReader focus on major patterns so that it does not need to create and execute HMM recognizers for the long tail of patterns that occur infrequently or happen to be exceptions to the general typeset text patterns of a book. Rejecting patterns does not mean, however, that the information in these rejected patterns will not be extracted. When we explain how ListReader constructs HMM recognizers in Section 5.3, we will also show that the HMMs constructed allow for deviations from the basic patterns for insertions, deletions, and substitutions, respectively like inserted birth times, missing days in dates, and OCR-error character substitutions in Figure 5.8. In essence, ListReader discovers the basic "clean" patterns in a book including a limited number of variations as a phrase structure grammar and also accommodates exceptions to basic patterns and larger variations with the HMM derived from that phrase structure grammar.

In grouping variations into basic clean patterns, ListReader's procedure can cluster field group templates that vary widely and therefore that a single linear HMM cannot reasonably accommodate. In Figure 5.6, for example, the four field group templates under the first "`[\n-Segment]`" are all quite different—varying in length (10, 5, 2, and 16 symbols, respectively) and content (respectively: given name and date, surname followed by given name, place, and two consecutive full names in two variations). To model widely differing patterns in a single high-level cluster, ListReader partitions patterns into similarity groups and for each group, selects a representative pattern as the basis for generating an HMM recognizer. We choose this approach of accounting for field group template variations instead of either creating a separate HMM recognizer for each variation or merging all the variations into a single union of parallel HMM recognizer. Using all possible variations is less desirable for two reasons: it significantly increases the running time of execution, which is quadratic in the number of states, and it significantly increases the variations that have to be hand-labeled. Merging all variations into a single pattern is probably hard to do without more domain knowledge than we expect the user to provide. Therefore, we choose to select a representative sample of field group variations from which ListReader builds its HMM recognizers.

In selecting field group template variations to be representatives, ListReader must be careful to observe significant differences and ignore minor variations. It therefore estimates the "distance" between variations and separates those that vary more than a pre-specified threshold and groups the rest with the separated templates such that they partition the space of field group template variations.

Then for each block of the partition, ListReader selects a “best” field template representative. From each representative ListReader generates a component of the HMM, as we explain in Section 5.3.

To measure the distance between pattern variations, ListReader uses a normalized Levenshtein edit distance: the minimum number of word insertions, deletions, and substitutions to transform one pattern to another divided by the average of the lengths of the patterns. Figure 5.9 shows the three pattern variations and their instances for the initial field group segment of the last record cluster in Figure 5.6. The normalized edit distance between the first two pattern variations, “`\n[UpLo],[Sp][UpLo]`” and “`\n[Up]'[UpLo],[Sp][UpLo]`”, is $2/((5+7)/2) = 0.333$—two insertions, “`[Up]`” and “`'`” into the first pattern (length 5), yields the second pattern (length 7). The normalized edit distance between the first and the third is 0.222, and between the second and third is 0.545. If the threshold is 0.5, the second and third pattern variations should be in separate groups. The first can be grouped with either the second or the third, and should be with the third in this example since it is “closer.”

```
[\n[UpLo],[Sp][UpLo]]           [\n[Up]'[UpLo],[Sp][UpLo]]
(count 20, length 5)            (count 1, length 7)
\nBarbor, Ninian                \nM'Gregor, John
\nBarbor, William
\nBarr, John
\nBlair, Hugh
\nCarruth, John
\nCochran, William              [\n,[Sp][UpLo]]
\nCraig, John                   (count 2, length 4)
\nErskine, John                 \n, William
\nInglis, John                  \n, Robert
\nKelloch, Mungo
\nLang, John
\nLochhead, John
\nLyle, John
\nMudie, John
\nOre, John
\nPaterson, John
\nShaw, John
\nSmith, David
\nWallace, John
\nWilson, Patrick
```

Figure 5.9: Initial Field Group Template Instances in the Last Record Cluster in Figure 5.6 Grouped by Pattern Variation.

When a group has more than one pattern variation, ListReader selects one to be the *representative* for the group. ListReader constructs its HMM from a combination of all the representatives, one for each group. To select the best representative, ListReader computes a representative score for a pattern variation by multiplying the pattern's length in symbols by its instance count and chooses the pattern variation with the highest score as the representative for the group. For the pattern variations in Figure 5.9 the scores are 20×5, 1×7, and 2×4 for the first, second, and third group respectively. Thus, for our example in Figure 5.9, "`\n[UpLo],[Sp][UpLo]`" is the representative for the group consisting of the first and third pattern and "`\n[Up]'[UpLo],[Sp][UpLo]`" is the representative of its own group. The representative score is motivated by observing that the score for a pattern is the number of tokens the pattern matches in the full text—more is likely to be better. In preliminary experiments on the *Shaver-Dougherty Genealogy*, ListReader's representative-selection procedure improved precision, recall, F-measure, and reduced the number of required hand-labelings compared to two other policies: one that selects the single longest pattern variation among all variations and one that selects the longest pattern for each group of pattern variations. Also, changing the edit distance cut-off values between 0.1 and 0.9 did not significantly affect the final evaluation scores, nor did normalizing by the length of only one of the compared patterns.

Figure 5.10 shows the representative templates along with an instances to illustrate them for the remaining field group template sequences in Figure 5.6. These template sequences also have some field-template variations, but perhaps different from what might be expected. The third cluster of records about couples and marriage dates does not have enough pattern variation to warrant breaking any of the field templates into groups. Date variations following the "`m.`" such as double-digit vs. single-digit days and abbreviated vs. non-abbreviated months are at most an edit distance or two apart (not enough when compared to the number of symbols in the "`[m-Segment]`" template). In addition, in these marriage records name patterns and patterns of record termination with no period are highly consistent across more than 100 pages in the *Kilbarchan Perish Record*. On the other hand, the first record cluster, with the template sequence "`[\n-Segment][\n-End-Segment]`", does break into several groups as Figure 5.10 shows.

Like the third record cluster, the second cluster's "`[\n-End-Segment]`" template breaks into two groups (normalized edit distance: 1/1.5 = 0.667), but neither of its two preceding field group templates has enough variation to cause a break into groups.

5.3 HMM Construction

An HMM is a probabilistic finite state machine consisting of a set of hidden states S, a set of possible observations W, an emission model $P(w|s)$ associating a state with a set of observable events, and a transition model $P(s_t|s_{t-1})$ associating one state with the next. States are initially "hidden" and must be inferred during application of the HMM from the observable events in the text. In our work, each event is a word-sized chunk of text (a token), including alphabetic words, numerals, spaces including newlines, and punctuation characters. Inferring the correct state associated with each word token is the main task done in extracting information from the text and is guided by the parameters of the HMM. Using the Viterbi algorithm, ListReader selects the most probable sequence of states given the words of the input text and the HMM's parameters. The emission model is a categorical distribution—a table of conditional probabilities indicating which observation w can be emitted[3] from which hidden state s and with what probability given s. The transition model is also a categorical distribution—a table of conditional probabilities indicating which hidden state s_t at position t can follow which other hidden state s_{t-1} at position $t-1$ and with what probability given s_{t-1}. The two kinds of probabilities are the parameters of the HMM. The set of states and the transitions that have non-zero probabilities in the transition model determine the structure of the state machine of the HMM. The processing described in Section 5.2 provides what ListReader needs to produce the set of hidden states and both the transition and the emission model for our application.

The HMM that ListReader constructs has two levels of structure, page-level and record-level, that are connected by transitions. The record-level states belong to record templates that are

[3] The term "emission" comes from the generative story commonly used to explain how an HMM can generate text. HMM parameters are traditionally chosen to maximize the likelihood that the HMM can generate the actual text that the HMM was meant to model.

```
[[\n-Segment][\n-End-Segment]]

        [\n-Segment]
            \n[UpLo],[Sp][DgDg][Sp][UpLo].[Sp][DgDgDgDg] :
                    \nJames, 15 Dec. 1672
            \n[UpLo],[Sp][UpLo] : \nAllasoun, Richard
            \n[UpLo] : \nLochwinnoch
            \n[Up]'[UpLo],[Sp][UpLo][Sp]?[Sp][UpLo][Sp][UpLo]-[Sp][Lo] :
                    \nM'Pherson, Mary ? Archibald Fer-\nguson

        [\n-End-Segment]
            .\n : .\n
            \n : \n

[[\n-Segment][born-Segment][\n-End-Segment]]

        [\n-Segment]
            \n[UpLo] : \nWilliam

        [born-Segment]
            ,[Sp][born][Sp][DgDg][Sp][UpLo].[Sp][DgDgDgDg] : , born 10 Dec. 1755

        [\n-End-Segment]
            .\n : .\n
            \n : \n

[[\n-Segment][and-Segment][m-Segment][\n-End-Segment]]

        [\n-Segment]
            \n[UpLo],[Sp][UpLo] : \nAiken, David

        [and-Segment]
            ,[Sp][and][Sp][UpLo][Sp][UpLo] : , and Janet Stevenson

        [m-Segment]
            [Sp][m].[Sp][DgDg][Sp][UpLo].[Sp][DgDgDgDg] :  m. 23 Nov. 1655

        [\n-End-Segment]
            \n : \n
```

Figure 5.10: Template Representatives of First, Second, and Third Record Cluster in Figure 5.6. (The "?" between "Mary" and "Archibald" is a character error; on the original page in the Kilbarchan book it is an m-dash.)

connected to each other and to the page-level states of the HMM. In Subsection 5.3.1, we explain how ListReader generates states for field group templates and how it labels states, providing them with syntactic and semantic IDs. (In Section 5.4 we tell how ListReader transforms these IDs into labels that map text associated with the HMM's hidden states to an application ontology.) In Subsection 5.3.2, we discuss transition and emission models for field group templates and say how ListReader sets parameters at the level of field group templates. Finally, in Subsection 5.3.3 we explain how ListReader finishes the transition model connecting HMM fragments for field group templates to each other to form HMM components for record templates and connecting record-template components to page-level HMM states and setting page-level transition and emission parameters.

5.3.1 Field Group Template State Generation

ListReader transforms each field group template representative into a linear sequence of HMM states, one HMM state for each word token in the parse tree of the field group segment. (In Figure 5.7 the dashed arrows show the sequence of word tokens considered for HMM construction.) Consider the representative templates in Figure 5.10 and particularly, for example, the representative template

```
[Sp]  [m]  .  [Sp]  [DgDg]  [Sp]  [UpLo]  .  [Sp]  [DgDgDgDg]
```

for the "`[m-Segment]`" (second from the bottom in Figure 5.10). The generated HMM fragment for this representative template has ten states, one for each word token in the representative template. Figure 5.11 shows these ten states. In addition to one state per word token in a field group template, ListReader generates an *insertion* state between every pair of consecutive word states. Figure 5.11 shows these insertion states as empty nodes. These insertion states allow for inconsistent punctuation and noise in pattern delimiters and for sparse comments such as when the time of birth is given as the first cluster in Figure 5.8 shows. Figure 5.11 also shows the transitions among the states for the representative template (albeit, as yet without the transition probabilities). The main transitions form a straight line through the template; the transitions to and from insertion states allow for text-addition deviations from a typical record; and the transitions that skip over word states allow

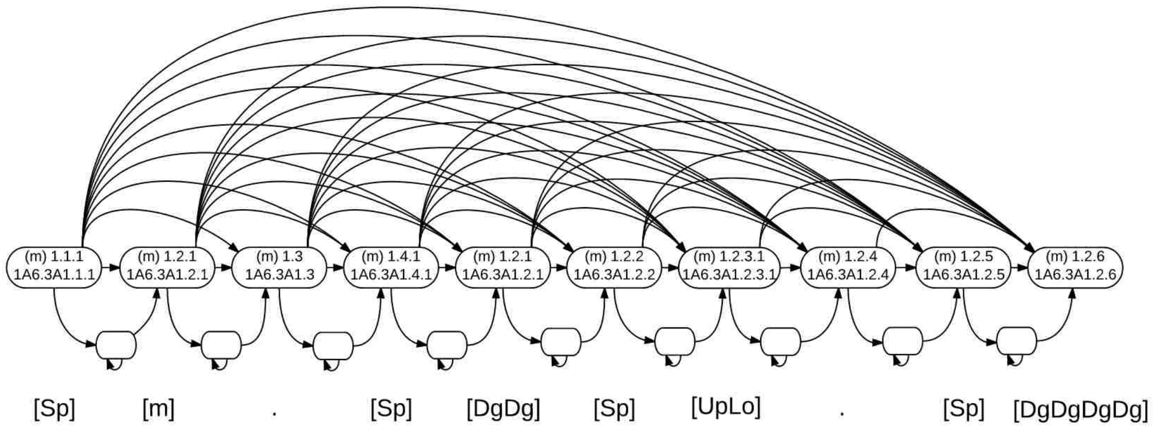


Figure 5.11: HMM States and Connecting Transitions for "`[m-Segment]`" whose Representative Template is "`[Sp][m].[Sp][DgDg][Sp][UpLo].[Sp][DgDgDgDg]`".

for text-omission anomalies. The missing days in the dates in the second cluster in Figure 5.8 is an example of when the HMM should skip states by taking a deletion transition. Patterns with inconsistencies and exceptions are those that tend to be discarded when ListReader culls its set of discovered patterns keeping only record patterns that are a clean sequence of representative field-group-template segments. Insertion states and deletion transitions help overcome deficiencies caused by inconsistencies and exceptions.

Figure 5.11 further shows that ListReader assigns each state two IDs—a *semantic ID* and a *syntactic ID*. Each state's syntactic ID (the second identifier in the nodes of Figure 5.11) is fundamentally a dot-delimited sequence of numbers representing the path in the parse tree from root (a record node that groups all segment nodes of a record template) to word token (the word token for which the state is being created). Each number in this path indicates the order of the node among its siblings in the parse tree. Consider, for example, the parse trees in Figure 5.7 with "`[Record]`" added as the root node forming one parse tree for the record template. Then the path to "`Aiken`" is

`[Record]`	`[\n-Segment]`	`[\n-FieldGroup]`	`[UpLo+]`	`[UpLo]`
`1`	`1`	`2`	`1`	`1`

and the path to "`1691`" is

[Record]	[m-Segment]	[m-FieldGroup]	[DgDgDgDg]
1	3	2	6

To make the syntactic ID of each HMM state functionally complete within the HMM, ListReader assigns “alternative” numbers (“A” numbers) to both the `[Record]`-level number and the `[...-Segment]`-level number in a path. The `[Record]`-level number identifies the alternative among all record templates discovered by ListReader, each of which implies a certain set of field group templates in a certain order. This record-level alternation number, along with the parse tree numbers, makes the syntactic ID of each node unique within the complete HMM. In our run of ListReader on the *Kilbarchan Parish Record*, the “`[[\n-Segment][and-Segment][m-Segment] [\n-End-Segment]]`” record happens to have been the 6th alternative record template found. The `[...-Segment]`-level number identifies which field-group alternative pertains to the state—1 in our example since the “`[m-Segment]`” in Figure 5.10 has only one alternative. Thus, in Figure 5.11 the tenth state, for example, has the syntactic ID “1A6.3A1.2.6”, with `[Record]`-level alternative “A6” and `[...-Segment]`-level alternative “A1”.

The semantic ID of an HMM state identifies the field group template to which the states of the field group apply and also the position of the token within the field group template. A semantic ID is the suffix of the syntactic ID starting with the field group segment’s alternation number. (We prepend the field group template’s identifying name parenthetically to the semantic ID for human readability. In Figure 5.11, since all the states belong to the “`[m-Segment]`” field template, they all begin with “(m)”.) The semantic ID, therefore, represents both a type of field group segment and a token’s position within that field group segment.

The semantic ID of a state is purposefully *not* unique within an HMM. States that share a semantic ID (and in turn the words they match) should be labeled the same because they have the same relationship with the primary object of their respective records. For example, all “m. <date>” constructs in the entire *Kilbarchan Perish Record* are marriage dates and should be labeled as such regardless of which record template in which they are found. (In our run of the Kilbarchan book, seven different record templates include the “`[m-Segment]`” field group template, two of which are in Figure 5.6.) ListReader should request only one label from the user for all states that share the same semantic ID—and therefore request only one label for the many hundreds of "m. <date>" strings in the Kilbarchan book. ListReader carefully infers semantic IDs and therefore carefully assigns field group alternation numbers. ListReader assigns the same field group alternation number to field group templates that (1) are of the same type (e.g. "`and`" and "`m`" are different) and (2) contain the same conflated field group words (e.g. "`[UpLo]`" and "`[UpLo][Sp][UpLo]`" are different). Therefore, HMM states with the same field group alternation number will have the same semantic ID if they are in the same position within their respective field group segments, even when those field groups appear in different record templates. This semantically ties the HMM states together that refer to the same field group templates and, in active sampling (Section 5.4.1), prevents the user from labeling more than one example of that field group. For example, the field group template "`[m-Segment]`" matching "` m. 29 Sept. 1691`" in the third cluster of Figure 5.6 also matches "` m. 18 Dec. 1672`" in the fourth cluster and will be assigned the same final labels because they will first be assigned the same semantic IDs, despite being in different positions in two different record templates.

5.3.2 Field Group Template Parameter Setting

ListReader sets the emission and transition parameters using maximum likelihood estimation (MLE). That is, they are set by normalizing the sums of counts of phrases in parse trees. These parameters must allow for flexible alignment of an induced HMM with text containing natural differences from the text on which the HMM is trained, such as word substitutions, insertions, and deletions. Beyond MLE, we also smooth these parameters using pseudo-counts (Dirichlet priors) to allow for combinations of events not present in the training data.

A substitution is a token in one record that does not exactly match the corresponding token in another record. For example, if an HMM fragment were built for the text "`\nJames, 15 Dec. 1672.\n`", we still expect the fragment to match text like "`\nJames, 15 Dee, 1672.\n`", despite the comma replacing the period and the "e" replacing the "c" (likely due to noise in the document causing OCR errors). ListReader allows for substitutions using both conflation of text and smoothing in the emission model: “`[UpLo]`” conflates “Dee” as well as “Dec”, and the emission model settings allow for alternatives in fixed text—punctuation and delimiter text. The emission model of each record-level state is set with the word-level conflated text for that state with a count of 1.0, unless the state is part of a delimiter in which case ListReader uses the non-conflated text, e.g. “`[m]`” instead of “`[Lo]`”. Emission parameters for conflated tokens belonging to numeral and alphabetic word character classes are smoothed with small, fractional pseudo-counts to allow for any other numeral or alphabetic words with low probability (lower for words outside of the character class of the original text). For example, in Figure 5.12 the word with conflated text “`[DgDgDgDg]`” receives a count of 1.0 for “`[DgDgDgDg]`”, a pseudo-count of 0.01 for “`[Dg]`”, “`[DgDg]`”, “`[DgDgDg]`”, and “`[DgDgDgDgDg]`” and a pseudo-count of 0.001 for “`[UpLo]`”, “`[LoUp]`”, “`[Up]`” and “`[Lo]`”. In general, ListReader’s construction of emission models promotes better alignment of similar words, especially words of the same character class, despite the small amount of training data provided and despite possible OCR errors and other variations. Similarly, as Figure 5.12 shows, ListReader adds pseudo-counts of spaces for the two kinds of internal space it encounters, (“ ” and “`\n`”), thus accommodating line breaks in the middle of a record where spaces usually appear. Finally, all record-level state emission models except for “`[Sp]`” and record delimiters (“`\n`” for our running example) receive a pseudo-count of 0.0001 for every other word in the document. We have omitted these smallest parameters from the figure for simplicity. After collecting the counts for a state’s emission model, ListReader sums the counts and divides the various counts by the sum to establish normalized emission probabilities that sum to one for each state’s emission model as Figure 5.12 shows.

A deletion is a sequence of one or more tokens of a record template that are missing in the text that should otherwise match that record template. For example, although the HMM fragment in Figure 5.11 is for text like “`\nJames, 15 Dec. 1672.\n`” we expect the HMM to be flexible enough to match text like “`\nJames 15 Dec. 1672\n`” (with some of the punctuation missing) or like “`\nJames, Dec. 1672.\n`” (with the day in the date missing).

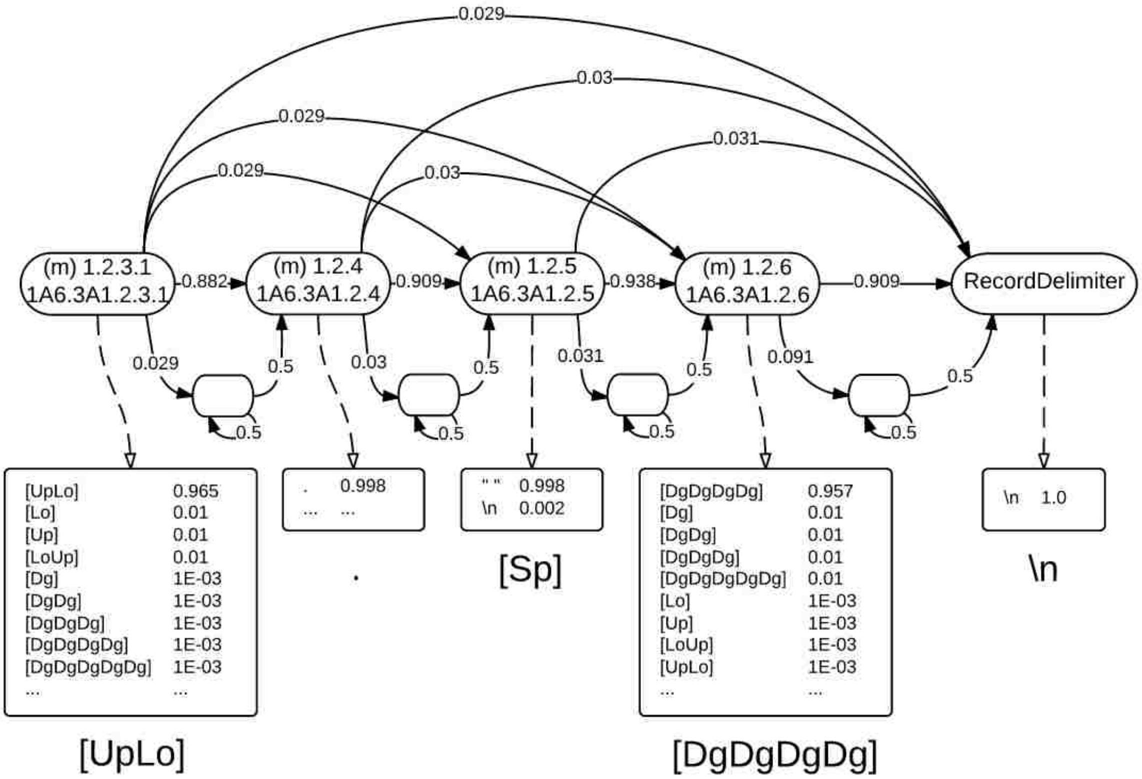


Figure 5.12: HMM Transition and Emission Models for the Last Four States of "`[m-Segment]`" and the Single State of "`[\n-End-Segment]`" for the Last Record Template in Figure 5.10.

To accommodate deletions, states in the HMM that are not adjacent in training data should become adjacent during execution. To allow for deletions during unsupervised training, the transition model of each pair of adjacent states receives a full count of 1.0, while the transition model of each pair of non-adjacent states receives a pseudo-count of $1/30$ if a pair of states satisfies our deletion constraint and zero otherwise. The deletion constraint requires an ordering on states: the second state must follow the first state within a training record, regardless of how far apart the words are. For example, in Figure 5.11 the "m" precedes the double-digit number for the day in the date, so the transition from the state representing "m" to the state representing the double-digit day receives the $1/30$ pseudo-count. But the reverse transition (from the double-digit state to the "m" state) would receive a zero pseudo-count. Proper order is determined algorithmically by comparing the parse tree numbers in the syntactic IDs of the two HMM states in question. The algorithm checks to see that the two states have ancestor numbers that are correctly ordered: e.g. states “1A6.3A1.1.3” for “m” and “1A6.3A1.2.1” for the double-digit state are correctly ordered because they have the same record alternation number (“A6”) and the same field-group-segment alternation number (“A1”) in the same position (3), and their field group parse tree numbers are in the correct order ($3.1.3 < 3.2.1$). But the reverse order would not be allowed. Field-group-segment alternation number can be different as long as their positions are also different. Figure 5.12 shows the probabilities on the deletion transitions (which also depend on out transitions to insertion states, discussed next).

An insertion is a sequence of one or more tokens appearing in text that should match a record template but which did not appear within the training text of that record template. For example, although the HMM fragment in Figure 5.11 is for text like “`\nJames, 15 Dec. 1672.\n`” we expect the HMM to be flexible enough to match text like “`\nJames, 15.  Dec.  1672..\n`” (containing two extra periods, perhaps because of noise in the text). It should also match text of occasional comments that may not have been seen in the training text, so that the HMM fragment for “`\nJames, 15 Dec. 1672.\n`” would also match the text “`\nJames, 15 Dec.  1672. (father dead)\n`” in which the author is adding a comment saying that a father died before the child’s birth. To accommodate as-yet unseen additions in a pattern, ListReader generates insertion states between record-level states s_{t-1} and s_t. Although not shown in our HMM figures, each insertion state has as its syntactic label the concatenation of the prior and subsequent states’ syntactic label ($s_t + s_{t-1}$) and as its semantic label the concatenation of the two states’ semantic labels. ListReader sets counts for three new transitions per insertion state: one to the insertion state from the prior state: s_{t-1} to ($s_t + s_{t-1}$), one self-transition for possible additional insertions: ($s_t + s_{t-1}$) to ($s_t + s_{t-1}$), and one from the insertion state to the subsequent state: ($s_t + s_{t-1}$) to s_t. The pseudo-count is the same for all three transitions: $n/30$, where n is 1, 2 or 3 depending on the likelihood of insertion at that location given prior knowledge of the behavior of insertions in list-like text: $n = 3$ next to record delimiters, $n = 2$ next to field group delimiters, and $n = 1$ everywhere else. In Figure 5.12, for example, the transition probability derived from the pseudo-count for the transition to the insertion state next to the record delimiter is greater than the probabilities on the transitions to other insertions states, which models the expectation that an insertion at the end of the record is more likely than in the middle.

5.3.3 Connecting the Pieces

As Figure 5.13 shows, ListReader generates page-level states for the beginning (*PageBeginning*) and ending (*PageEnding*) of each page and connects them to states for non-list text (*NonList*) and for list-record text (*RecordDelimiter*), the beginning state for all record-template HMMs. Figure 5.13 also shows how ListReader connects its record-delimiter state to every HMM record template—all 47 of them for our example run of the *Kilbarchen Parish Record*. One of the record-template HMMs is open, schematically showing the interconnections of the HMM fragments for

```
[\n-Segment][born-Segment][\n-End-Segment]
```

the second record template in Figure 5.10. Notice that the field group template "`[\n-End-Segment]`" has two representative templates, one for "`.\n`" and one for "`\n`". Whenever a field group template has multiple representative templates, ListReader generates parallel HMM fragments, one for each identified representative template. Each field group template has an HMM fragment of the form of the field-template HMM for "`[m-Segment]`" in Figure 5.11. Schematically, Figure 5.13 shows for each HMM field group template only the initial and final states along with the entry and exit insertion states. Each HMM fragment requires connections to all prior and subsequent HMM fragments as Figure 5.13 shows. When consecutive parallel HMM fragments occur as they would for the first record template in Figure 5.10, the HMM fragments are connected in a cross-product fashion. Thus ListReader would generate sixteen transitions to connect the pieces of the first record template in Figure 5.10, which contains four HMM fragments for "`[\n-Segment]`". Each of the four representative field group templates has four out-transitions, three going to the two end-record HMM fragments and one going directly to the record delimiter as in Figure 5.13.

ListReader creates transitions from the *RecordDelimiter* state to the start states of each of the initial field group templates for every record template it discovers. ListReader sets the count for each

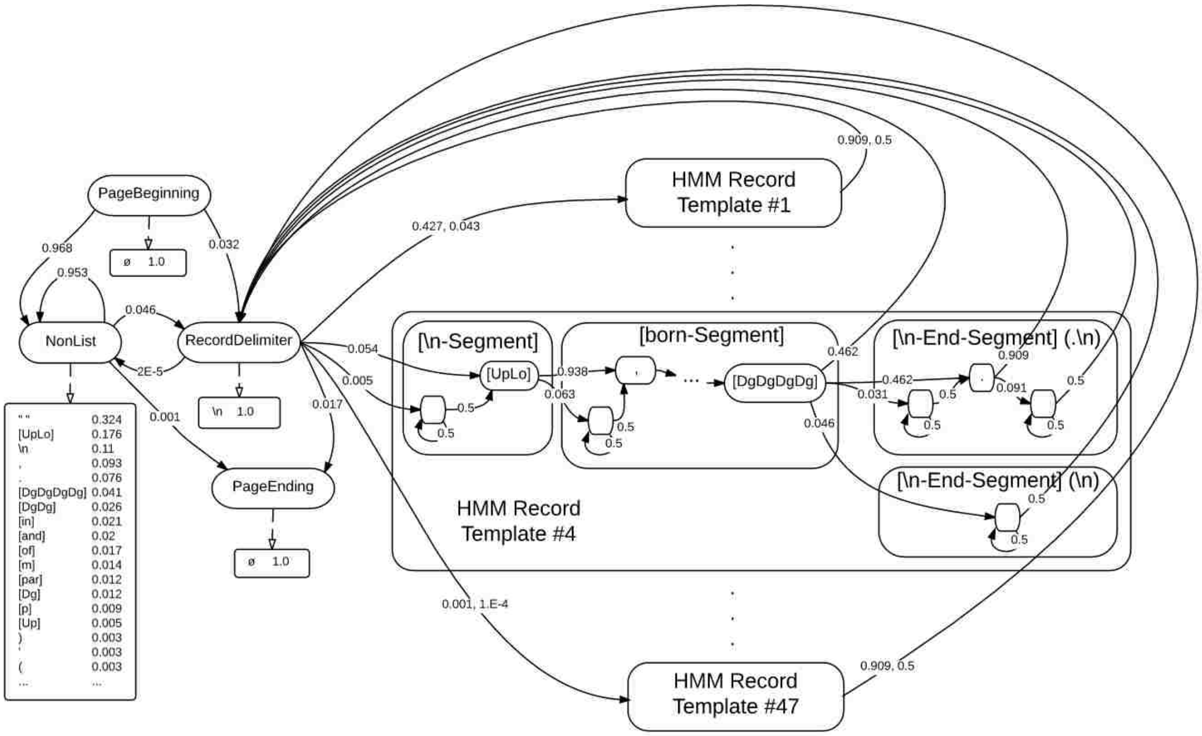


Figure 5.13: Schematic Diagram of ListReader-generated HMM.

of these transitions equal to the sum of the sizes of the clusters in the group for the representative template being modeled. For the HMM of the record template for the group consisting of the first and third clusters in Figure 5.6, for example, the count would be 22 (20 for the first cluster plus 2 for the third). The second cluster in Figure 5.6 is in a group of its own and has the count 1. Counts for transitions from the *RecordDelimiter* state to an initial insertion state are $3/30$ as specified earlier for an insertion state adjacent to a record delimiter. The count for the out-transition to the *NonList* state is the number of locations in the book where a record delimiter is followed by text that was not identified as being part of a list record, and the count for the out-transition to the *PageEnding* state is the number of pages in the book that end in a record delimiter. Summing these counts and normalizing them yields the transition probabilities for transitions emanating from the *RecordDelimiter* state. Figure 5.13 shows eight of these transition probabilities for our example run of the *Kilbarchen Parish Record*, with two of the edges in the figure contain parameters for both a transition to an insertion state and to a non-insertion state separated by a comma.

ListReader generates states and transitions for its page-level model as Figure 5.13 shows. The emission model of *PageBeginning* and *PageEnding* are fixed to contain only the special character that ListReader artificially inserts into the text sequence at the beginning and ending of each page to represent page breaks. The emission model of *RecordDelimiter* is fixed to contain the set of allowable record delimiters, which currently contains only the newline character. For these fixed emission models, the probability of the allowable character is 1.0 and all other probabilities are 0.0. The emission model of *NonList* state is not fixed. Rather, it is set as the MLE estimate of all word tokens in the input text that were not covered by any candidate records during unsupervised grammar induction. The emission model for the *NonList* state in Figure 5.13 lists several of these word tokens and their probabilities based on actual occurrence counts in our run of ListReader on the *Kilbarchan Parish Record*. The most frequently occurring word token not included in candidate records is the space character with 32.4% of the uncovered tokens and "`[UpLo]`" with 17.6%. Three of many tokens that appear only once are "mile", "are", and "about"—not shown in Figure 5.13 along with many others. We train the emission model of the *NonList* state on unlabeled data and the emission models of list states on labeled data (specifically automatically-labeled data). These two sets of states (list and non-list) can be seen as a binary classifier, predicting a "positive" and a "negative" class. We justify our approach to training our HMM from mixed labeled and unlabeled data by citing Elkan and Noto ([21]) who show that for binary classifiers, "under the assumption that the labeled examples are selected randomly from the positive examples ... a classifier trained on positive and unlabeled examples predicts probabilities that differ by only a constant factor from the true conditional probabilities of being positive." We also smooth the emission model of the *NonList* state using small Dirichlet priors to allow any word to appear, even those not appearing in the training data.

The parameters for the other transitions among the four page-level states are also trained using MLE from the records discovered during grammar induction. For the emanating transitions of the *PageBeginning* state, for example, if there were 100 pages of input text and 10 of the pages began with list text and 90 with non-list text, then the transition from *PageBeginning* to *RecordDelimiter* would receive a count of 10 while the transition from *PageBeginning* to *NonList* would receive a count of 90. The transition model is also smoothed with small Dirichlet priors to allow any reasonable transitions that were not seen in the parsed text such as a transition from *PageBeginning* to *PageEnding*, allowing for an empty page or a page consisting only of a picture. For the *NonList* state, the count for the transition to the *RecordDelimiter* state is the number of instances in the book where ListReader identified text outside a list immediately before a list; the count for the self loop returning to the *NonList* state is number of word tokens in the book that are not covered by record templates and do not immediately precede a list; and the count for the transition to the *PageEnding* state is the number of pages in the book that do not end with a list. ListReader normalizes counts for each emanating state, producing the transition probabilities.

Figure 5.13 shows the transition probabilities for our example run of the *Kilbarchan Perish Record*. Altogether, the full ListReader-generated HMM for our example run of the *Kilbarchan Parish Record* has 1,805 states and 3,717 transitions.

5.4 Labeling and Final Extraction

To populate an ontology with extracted information, ListReader (1) obtains labels from a user for HMM states and (2) maps labeled text to the ontology. To obtain labels, ListReader actively and selectively requests labels that associate HMM states with elements of the ontology, as explained in Subsection 5.4.1. ListReader then applies obtained state-label knowledge to extract information from throughout the input text and map it to the ontology, as explained in Subsection 5.4.2.

5.4.1 Active Sampling

Active sampling consists of a cycle of repeated interaction with the user. On each iteration of the loop, ListReader selects and highlights text that matches part of the HMM, and the user labels the fields in highlighted text. Labeling consists of the user copying substrings of the ListReader-selected text into the entry fields of the data entry form in ListReader's UI as Figure 5.1 shows. ListReader then accepts the labeled text via the web form interface and assigns labels to the corresponding

HMM states, which completes that part of the HMM and enables it to become a “wrapper” that extracts information from the text and maps it to the ontology as we explain in Subsection 5.4.2.

The active sampling cycle is a modified form of active learning, focusing on the “active sampling” step and performing practically none of the “model update” step, just as in [38]. The HMM training ListReader does is fully unsupervised—no HMM structure or parameter learning takes place under the supervision of a user either interactively or in advance. Label renaming is the only change ListReader makes to the HMM during active sampling. In each cycle, ListReader actively selects the text for labeling that maximizes the return for the labeling effort expended. To initialize the active sampling cycle, ListReader applies the HMM to the text of each page in the book. It labels the strings that match each state with the state’s semantic ID. ListReader saves the count of matching strings for each semantic ID. It also records the page and character offsets of the matching strings throughout the book and their associated semantic IDs. ListReader uses the page and character offsets when highlighting a span of text in the UI for the user to label. ListReader selects a span of text on each iteration of active sampling using a query policy (explained next) that is based on the counts of matching strings for each semantic ID.

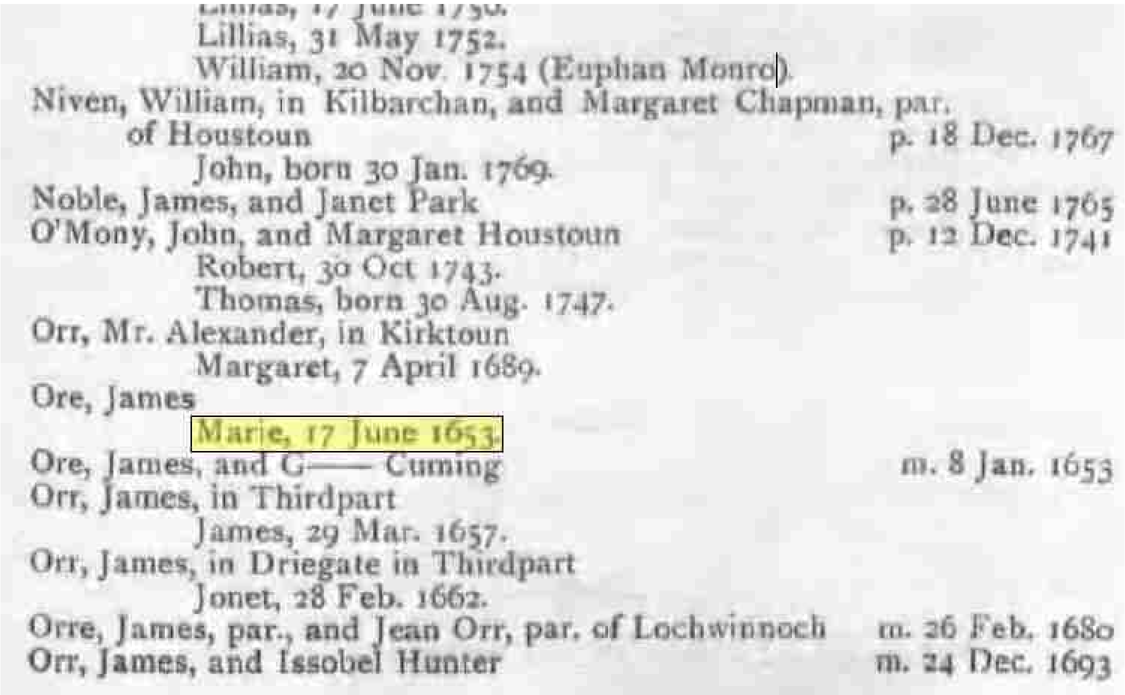
The string ListReader selects as “best” is a string that matches the HMM fragment with the highest predicted return on investment (ROI). The ROI can be thought of as the slope of the learning curve: higher accuracy and lower cost produce higher ROI. The HMM fragments considered are HMM record templates or contiguous parts thereof (e.g. the HMM fragment for “`[m-Segment]`” illustrated in Figure 5.11). When more than one string matches the best HMM fragment, ListReader selects the first one on whichever page contains the most matches of that HMM fragment. ListReader computes the predicted ROI as the sum of the counts of the strings matching each state in the candidate HMM fragment divided by the number of states in the HMM fragment—that is, the average match-count per state. Querying the user to maximize the immediate ROI tends to maximize the slope of the learning curve and has proven effective in other active learning situations [33]. Once the user labels the selected text, ListReader removes the counts for all strings that match the corresponding states or that share the semantic IDs of labeled states, recomputes the ROI scores of remaining states, and issues another query to the user.

In our example run of the *Kilbarchan Parish Record*, ListReader selects the highlighted text in Figure 5.14. Its HMM record template is composed of the first representative for the first “`[\n-Segment]`” field group template and the first representative for the first “`[\n-End-Segment]`” field group template in Figure 5.10. There are nine matching states in this HMM record template, one for each word-level, non-record-delimiter symbol, “`[UpLo],[Sp][DgDg][Sp][UpLo][Sp][DgDgDgDg].`”. The hit count for the strings matching each state are:

`[UpLo]`	2680
`,`	2678
`[Sp]`	2691
`[DgDg]`	2680
`[Sp]`	2678
`[UpLo]`	2679
`[Sp]`	2682
`[DgDgDgDg]`	2683
`.`	3840

whose sum is 25,291 and whose ROI score is thus $25291/9$ and is greater than the ROI score for any other HMM record template. Intuitively, this makes sense because the most often occurring fact assertion in the *Kilbarchan Parish Record* is statement about a christening of a child of the form “<GivenName>, <Day> <Month> <Year>”, of which there are thousands.

When one HMM state receives a user-supplied label, all states sharing the same semantic ID receive the same final label. In the example in Figure 5.14 the user would label “Marie” as *KilbarchanPerson.Name.GivenName*, “17” as *KilbarchanPerson.ChristeningDate.Day*, “June” as *KilbarchanPerson.ChristeningDate.Month*, and “1653” as *KilbarchanPerson.ChristeningDate.Year*. And, since given-name and date fields in other christening record-templates have the same semantic

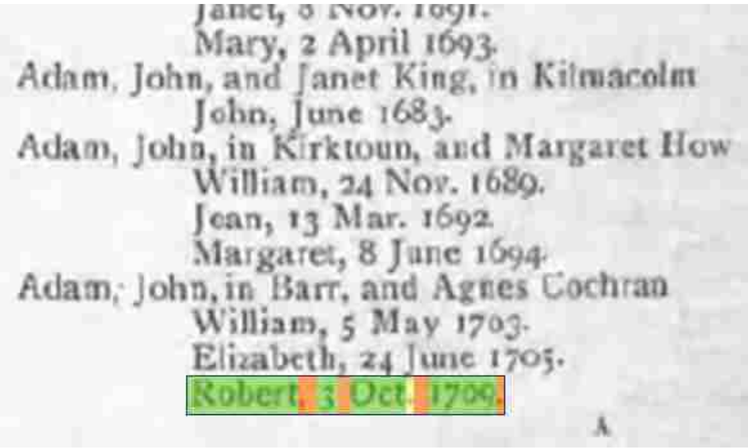


Lillias, 17 June 1750.
Lillias, 31 May 1752.
William, 20 Nov. 1754 (Euphan Monro).
Niven, William, in Kilbarchan, and Margaret Chapman, par.
of Houstoun p. 18 Dec. 1767
John, born 30 Jan. 1769.
Noble, James, and Janet Park p. 28 June 1765
O'Mony, John, and Margaret Houstoun p. 12 Dec. 1741
Robert, 30 Oct 1743.
Thomas, born 30 Aug. 1747.
Orr, Mr. Alexander, in Kirktoun
Margaret, 7 April 1689.
Ore, James
Marie, 17 June 1653.
Ore, James, and G—— Cuming m. 8 Jan. 1653
Orr, James, in Thirdpart
James, 29 Mar. 1657.
Orr, James, in Driegate in Thirdpart
Jonet, 28 Feb. 1662.
Orre, James, par., and Jean Orr, par. of Lochwinnoch m. 26 Feb. 1680
Orr, James, and Issobel Hunter m. 24 Dec. 1693

Figure 5.14: First Active-Sampling User Query.

IDs, these fields are also labeled—thousands of them due to the date variations (abbreviated/non-abbreviated months and single-digit/double-digit days) that appear in the *Kilbarchan Perish Record*. Furthermore, all delimiters are implicitly labeled whenever a user labels the fields in a record as the text between labeled fields and preceding the first labeled field and following the last labeled field. In the example in Figure 5.14, the user implicitly labels four delimiters: the comma and space between the name and the day in the date, the two spaces within the date, and the period following the year. The states for delimiters also have semantic IDs as Figure 5.11 shows, so ListReader propagates the labels to all other states with identical semantic IDs—those that have the same delimiter in the same position in the same field group template.

ListReader's label propogation across semantic IDs minimizes the user's labeling effort during active sampling. As an example, Figure 5.15 shows ListReader's second active-sampling query for our example run of the *Kilbarchan Parish Record*. The highlighting is multicolored: green for previously labeled fields (the *GivenName* "Robert", the *ChristeningDate.Day* "3", the *ChristeningMonth* "Oct", and the *Christening.Year* "1709"); red for previously labeled delimiters



Janet, 8 Nov. 1691.
Mary, 2 April 1693.
Adam, John, and Janet King, in Kilmacolm
John, June 1683.
Adam, John, in Kirktoun, and Margaret How
William, 24 Nov. 1689.
Jean, 13 Mar. 1692.
Margaret, 8 June 1694.
Adam, John, in Barr, and Agnes Cochran
William, 5 May 1703.
Elizabeth, 24 June 1705.
Robert, 3 Oct. 1709.
A

Figure 5.15: First Active-Sampling User Query Requiring Only Partial Labeling.

(“, ”, “ ”, “ ”, and “.”); and yellow for unlabeled text (the period following “Oct” in Figure 5.15). ListReader does not know, by what it has so far learned, whether the period following “Oct” belongs to the *Month* field or to the delimiter between “Oct” and “1709”. At this point, the user should copy “Oct.” into the *KilbarchanPerson.Christening.Month* form field to label the “`.`” state following the “`[UpLo]`” state now known as the *KilbarchanPerson.ChristeningDate.Month* state in the HMM fragment as being part of the month field.

As our Kilbarchan example shows, active sampling is impactful from the first query. Furthermore, it improves recall monotonically as it does not back-track or reverse labeling decisions from one cycle to the next. Compared with typical active learning [53], it is not necessary for ListReader to induce an intermediate model from labeled data before it can become effective at issuing queries. This would be true even if ListReader did update the HMM during active learning cycles, although it would necessitate ListReader having to apply the HMM again on every cycle, which currently it avoids. Furthermore, ListReader need not know all the labels at the time of the first query. Indeed, it starts active sampling without knowing any labels. The query policy is similar to processes of novelty detection [45] in that it effectively identifies new structures for which a label is most likely unknown. Furthermore, the wrapper can be induced for complete records regardless of how much the user annotates or wants extracted, and ListReader is not dependent on the user to identify record- or field-delimiters nor to label any field the user does not want to be extracted.

5.4.2 Mapping Data to Ontology

Having completed the HMM wrapper, including user-supplied labels, ListReader applies the HMM using the Viterbi algorithm a second time to compute the most probable sequence of state IDs for each token in each page, translates the syntactic IDs into user-supplied labels for each token, and then translates text strings labeled with form-field labels into predicates that it inserts into the ontology. The entire flow from HTML form and text (e.g. Figure 5.1) to ontology (e.g. Figure 5.2) takes a few steps, as we now explain. To automate much of this process, we have established formal mappings among three types of knowledge representation: (1) HTML forms (e.g. Figure 5.1), (2) ontology structure (e.g. Figure 5.2), and (3) in-line labeled text (e.g. Figure 5.3). These mappings effectively reduce the ontology population problem to a sequence labeling problem, and in turn the sequence labeling problem to a form-construction and form-filling task, a process more familiar to most users than either sequence labeling or ontology population.

The mapping begins with the user-constructed HTML form. The structure of the form is a tree of nested, labeled form fields. The names of some of the form fields may be the same, in which case they will map to the same object set in the ontology, resulting in a non-tree shaped ontology. The leaves of the tree of form fields are lexical text-entry fields into which the user inserts field text from the page by clicking on the text. ListReader maps form fields to object sets (concepts or unary predicates) and uses the nesting of one field inside another to produce a relationship set (n-ary predicates $n > 1$) among object sets. The root of the tree is the form title and represents the primary object set, i.e. the topical concept of a record in a list, for example *KilbarchanPerson* in Figure 5.1.

ListReader maps the empty HTML form to an ontology schema that may contain a number of conceptual distinctions including any of the following. (1) textual vs. abstract entities (e.g. *GivenName*("Archibald") vs. *KilbarchanPerson*($Person_1$) in Figure 5.1, where $Person_1$ is an object identifier); (2) 1-many relationships in addition to many-1 relationships so that a single object can relate to many associated entities or only one (e.g. a *KilbarchanPerson* object in Figure 5.2 can relate to several *Parish*es but only one *ChristeningDate*—the arrowhead in the diagram on *ChristeningDate* designating functional, only one, and the absence of an arrowhead on *Parish* designating non-functional, allowing many); (3) n-ary relationships among two or more entities instead of strictly binary relationships (e.g. if a user wants to associate dates of residence in a parish along with the parish name yielding a ternary relationship among *KilbarchanPerson*, *Parish*, and *ResidenceDates* in Figure 5.2 and being designated by a double-column multiple-entry field with *ResidenceDates* along side of *Parish* in Figure 5.1); (4) ontology graphs with arbitrary path lengths from the root instead of strictly unit-length as in named entity recognition or data slot filling (e.g. *KilbarchanPerson.Spouse.MarriageDate.Day* in Figure 5.2); (5) concept categorization hierarchies, including, in particular, role designations (e.g. if a user wants to designate roles for some Kilbarchen persons who have duties in the parish, such as a priest or an alter boy); and (6) a non-tree ontology structure (object sets can be shared among multiple relationship sets). This expressiveness provides for the rich kinds of fact assertions we wish to extract in our application.

After active sampling is complete, ListReader labels the text of each page as illustrated in Figure 5.3 and translates the labeled text into predicates and inserts them into the ontology. Record delimiter tags surround a complete record string and determine which fields belong to the same record. ListReader splits labels into object set names and instantiates objects for each new object set name and relationship predicates for each dot-separated sequence of object set names. The text string of the each leaf field is instantiated as a lexical object. Any remaining unlabeled text (text labeled as *NonList* or text still labeled with its original semantic ID) produces no output.

5.5 Evaluation

We evaluate ListReader on two books, the *Shaver-Dougherty Genealogy* and the *Kilbarchan Parish Register*, and compare its performance to two baselines, an implementation of the Conditional Random Field (CRF) and a previous version of ListReader that induced regular-expression wrappers instead of HMM wrappers [49]. The regex version of ListReader is similar to the HMM version except that it creates separate regular expression wrappers for every record pattern discovered during grammar induction whereas the HMM version is selective about which record and field group templates make it into the final HMM wrapper. The motivation for creating the HMM version is to overcome the brittleness of regular expressions, believing that the more malleable HMM wrappers would yield better recall results because of their ability to recognize variations in text patterns without requiring an exact match and would not hurt precision results too much because of ListReader's ability to create HMMs with a high degree of correlation to the observed text.

In Subsection 5.5.1, we describe the data (books) we used to evaluate ListReader. We explain the experimental procedure for evaluating the CRF in Subsection 5.5.2. We give the metrics we used in Subsection 5.5.3 and the results of the evaluation in Subsection 5.5.4, which includes a statistically significant improvement in F-measure as a function of labeling cost.

5.5.1 Data

General wrapper induction for lists in noisy OCR text is a novel application with no standard evaluation data available and no directly comparable approaches other than our own previous work. We produced development and evaluation data for the current research from three separate family history books.[4]

We developed ListReader almost entirely using the text of the *The Ely Ancestry* [6] and *Shaver-Dougherty Genealogy* [54]. *The Ely Ancestry* contains 830 pages and 572,645 word tokens and *Shaver-Dougherty Genealogy* contains 498 pages and 468,919 words. We used *Shaver-Dougherty Genealogy* and three pages of the *Kilbarchan Parish Register* [26] containing 6013 words as our evaluation data. The *Kilbarchan Parish Register* would be considered a blind test except for our recognizing the need to not conflate lower-case words for this kind of book. We have added this option as an input parameter that is easy to set after quickly inspecting the input document. We chose the two test books to represent larger and more complex text on the one hand using the *Shaver-Dougherty Genealogy* and smaller and simpler text on the other using the *Kilbarchan Parish Register*.

[4] We will make all text and annotations available to others upon request.

The *Kilbarchan Parish Register* is a book composed mostly of a list of marriages and sub-lists of children under each marriage. The three pages we used as our test set are in the Appendix.

To label the text, we built a form in the ListReader web interface, like the one on the left side of Figure 5.1 that contains most of the information about a person visible in the lists of selected pages. Using the tool, we selected and labeled all the field strings in 68 pages from *Shaver-Dougherty Genealogy* and 3 pages from the *Kilbarchan Parish Register*. We ran the unsupervised wrapper induction on the text of the labeled pages. Grammar induction did not use the labels, but active sampling used a small number of them, namely those for the text selected by ListReader during active sampling. All of the remaining labels were used as ground truth for evaluation. The web form tool generated and populated the corresponding ontologies which we used as the source of labeled text. The annotated text from the 68 pages of the *Shaver-Dougherty Genealogy* have the following statistics: 14,314 labeled word tokens, 13,748 labeled field instances, 2,516 record instances, and 46 field types. Figure 5.16 shows the two ontologies used for these 46 field labels—one for the main body of the paper and one for the index. The annotated text from the 3 pages of the *Kilbarchan Parish Register* have the following statistics: 852 labeled word tokens, 768 labeled field instances, 165 record instances, and 12 field types. Figure 5.2 shows the ontology corresponding to those 12 field labels—the 12 paths from the *KilbarchanPerson* object set to the leaf lexical object sets after combining *MarriageDate* with *ProclamationDate* and combining *BirthDate* with *ChristeningDate*.

5.5.2 CRF Comparison System

We believe the performance of the supervised Conditional Random Field (CRF) serves as a good baseline or reference point for interpreting the performance of ListReader. The CRF implementation we applied is from the Mallet library [46]. To ensure a strong baseline, we performed feature engineering work to select an appropriate set of word token features that allowed the CRF to perform well on development test data. The features we applied to each word include the case-sensitive text of the word, and the following dictionary/regex Boolean attributes: given name dictionary (8,428

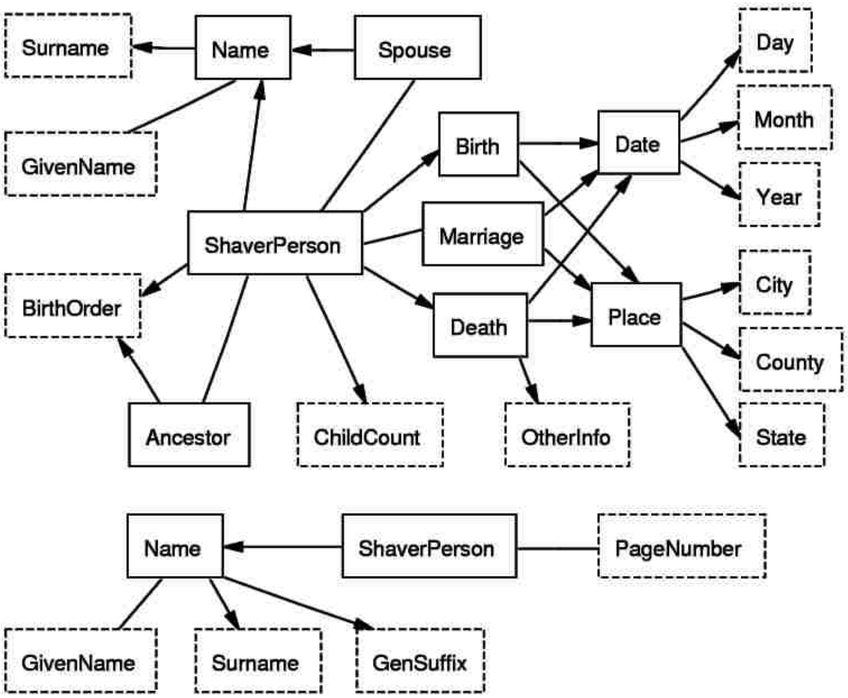


Figure 5.16: ShaverPerson Ontologies.

instances), surname dictionary (142,030 instances), names of months (25 variations), numeral regular expression, roman numeral regular expression, and name initial regular expression (a capital letter followed by a period). The name dictionaries are large and have good coverage of the names in the documents. We also distributed the full set of word features to the immediate left and right neighbors of each word token (after appending a "left neighbor" or "right neighbor" designation to the feature value) to provide the CRF with contextual clues. Using a larger neighbor window than just right and left neighbor did not improve its performance. These features constitute a greater amount of knowledge engineering than we allow for ListReader. We simulated active learning of a CRF using a random sampling strategy—considered to be a hard baseline to beat in active learning research, especially early in the learning process [14].

Each time we executed the CRF, we trained it on a random sample of n lines of text sampled throughout the hand-labeled portion of the corpus. Then we executed the trained CRF on all remaining hand-labeled text. We varied the value of n from 1 to 10 to fill in a complete learning curve. We ran the CRF 7,300 times for the *Shaver-Dougherty Genealogy* and 4,000 times for the

Kilbarchan Parish Register and then computed the average y value (precision, recall, or F-measure) for each x value (cost) along the learning curve and generated a locally weighted regression curve from all 7,300 (or 4,000) points.

5.5.3 Experimental Procedure and Metrics

To test the three extractors (two versions of ListReader and the CRF) we wrote an evaluation system that automatically executes active sampling by each extractor, simulates manual labeling, and completes the active sampling cycle by reading in labels for ListReader and by retraining and re-executing the CRF. The extractors incur costs during the labeling phase of each evaluation run which includes all active sampling cycles up to a predetermined budget. To simulate active sampling, the evaluation system takes a query from the extractor and the manually annotated portion of the corpus and then returns just the labels for the text specified by the query in the same way the ListReader user interface would have. In this way, we were able to easily simulate many active sampling cycles within many evaluation runs for each extractor.

For purposes of comparison, we computed the accuracy and cost for each evaluation run. We measured cost as the number of field labels provided during the labeling phase, a count that correlates well with the amount of time it would take a human user to provide the labels requested by active sampling. The CRF sometimes asks the user to label prose text while ListReader does not. To be consistent in measuring cost, we do not count these labelings against the cost for the CRF. This means that the CRF has a slight advantage as it receives training data for negative examples (prose text) without affecting its measured cost. During the test phase, the evaluation system measured the accuracy of the extractors only on tokens of text not labeled for active sampling.

Since our aim is to develop a system that accurately extracts information at a low cost to the user, our evaluation centers on a standard metric in active learning research that combines both accuracy and cost into a single measurement: Area under the Learning Curve (ALC) [14]. The rationale is that there is no single, fixed level of cost that is right for all information extraction projects. Therefore, the ALC metric gives an average learning accuracy over many possible budgets.

We primarily use F_1-measure as our measure of extraction accuracy, although we also report ALC for precision and recall curves. Precision is defined to be $\frac{tp}{tp+fp}$ and recall is defined to be $\frac{tp}{tp+fn}$ where *tp* means true positive, *fp* means false positive, and *fn* means false negative field strings. F-measure (F_1) is the harmonic mean of precision (p) and recall (r), or $\frac{2pr}{p+r}$. ALC is $\int_{min}^{max} f(c)dc$, where c is the number of user-labeled fields (cost) and $f(c)$ can be precision, recall, or F-measure as a function of cost, and *min* and *max* refer to the smallest and largest numbers of hand-labeled fields in the learning curve. The curve of interest for an extractor is the set of an extractor's accuracies plotted as a function of their respective costs. The ALC is the percentage of the area, between 0% and 100% accuracy and *min* and *max* cost, that is covered by the extractor's accuracy curve. ALC is equivalent to taking the mean of the accuracy metric at all points along the curve over the cost domain—an integral that is generally computed for discrete values using the Trapezoidal Rule,[5] which is how we compute it.

5.5.4 Results

From Tables 5.1 and 5.2 we see that the ALC of F-measure for ListReader (HMM) is significantly higher than that of ListReader (Regex) for both books, which in turn is significantly higher than that of the CRF. ListReader (HMM) consistently outperforms the CRF in terms of F-measure over both learning curves. ListReader (Regex) consistently produces only a few false positives (precision errors). The improvement of ListReader (HMM) over (Regex) is due to improved recall. The ListReader-generated HMM is capable of recognizing up to almost 50% more list records in the input text document than the phrase structure grammar from which it is built, despite the fact that HMM construction eliminates between about 50% and 90% of the patterns found in the second suffix tree to satisfy our record selection constraints while the Regex preserves all of them. ListReader (HMM) does not produce as high a precision as ListReader (Regex), but does improve on recall. Recall is improved because the HMM matches more records with fewer record templates

[5] See `http://en.wikipedia.org/wiki/Trapezoidal_rule`

Table 5.1: ALC of Precision, Recall, F-measure for the *Shaver-Dougherty Genealogy* (%).

	Prec.	Rec.	F_1
CRF	50.63	33.95	38.82
ListReader (Regex)	**97.60**	32.55	48.78
ListReader (HMM)	69.59	**42.84**	**52.54**

All differences are statistically significant at $p < 0.05$ using an unpaired t test except for the difference in Recall of ListReader (Regex) and the CRF.

Table 5.2: ALC of Precision, Recall, F-measure for the *Kilbarchan Parish Register* (%).

	Prec.	Rec.	F_1
CRF	68.86	63.02	65.47
ListReader (Regex)	**96.34**	54.30	67.92
ListReader (HMM)	**91.38**	**72.74**	**79.19**

All differences are statistically significant at $p < 0.05$ using an unpaired t test except for the difference in Precision of the two ListReaders and the difference in Recall of ListReader (Regex) and the CRF.

on account of its flexible probabilistic structure, allowing the user to provide fewer labels to cover more information (allowing the HMM to reach the end of the long tail of record templates faster).

From Table 5.2 we see that in the *Kilbarchan Parish Register*, ListReader (HMM) outperforms ListReader (Regex) and the CRF in all three metrics except in the case of Regex's precision, but that difference is not statistically significant as it is in *Shaver-Dougherty Genealogy*.

Figures 5.17, 5.18, and 5.19 show plots of the F-measure, precision, and recall learning curves for ListReader and the CRF on the *Shaver-Dougherty Genealogy* and Figures 5.20, 5.21, and 5.22 show plots of the F-measure, precision, and recall learning curves for ListReader and the CRF on the *Kilbarchan Parish Register*. These plots provide detail behind the ALC metrics in Tables 5.1 and 5.2. Visually, the learning curves indicate that ListReader (Regex and HMM) both outperform the CRF fairly consistently over varying numbers of field labels for all three metrics. Tables 5.1 and 5.2 tell us that the differences among the three extractors are statistically significant for most pairwise comparisons at $p < 0.05$ using an unpaired t test. The three pairs that are not significant are the ones comparing the recall of ListReader (Regex) and the CRF on both the *Shaver-Dougherty Genealogy* and the *Kilbarchan Parish Register* and comparing the precision of the two versions of ListReader on the *Kilbarchan Parish Register*.

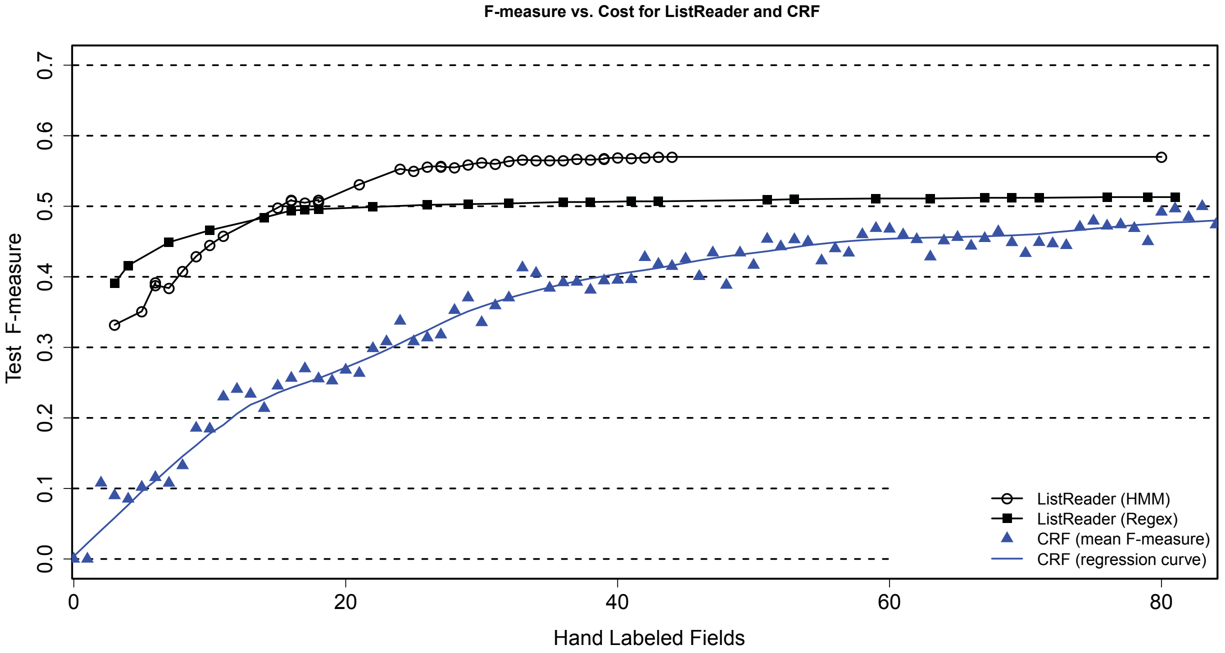


Figure 5.17: F-measure Learning Curves for the *Shaver-Dougherty Genealogy*.

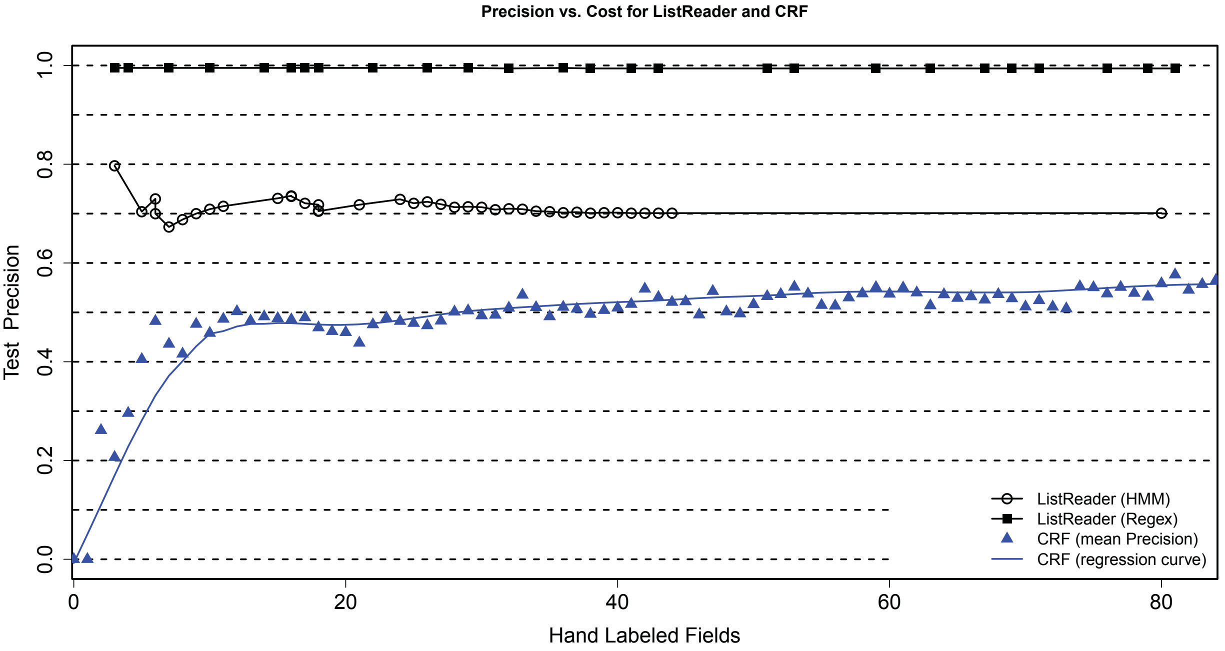


Figure 5.18: Precision Learning Curves for the *Shaver-Dougherty Genealogy*.

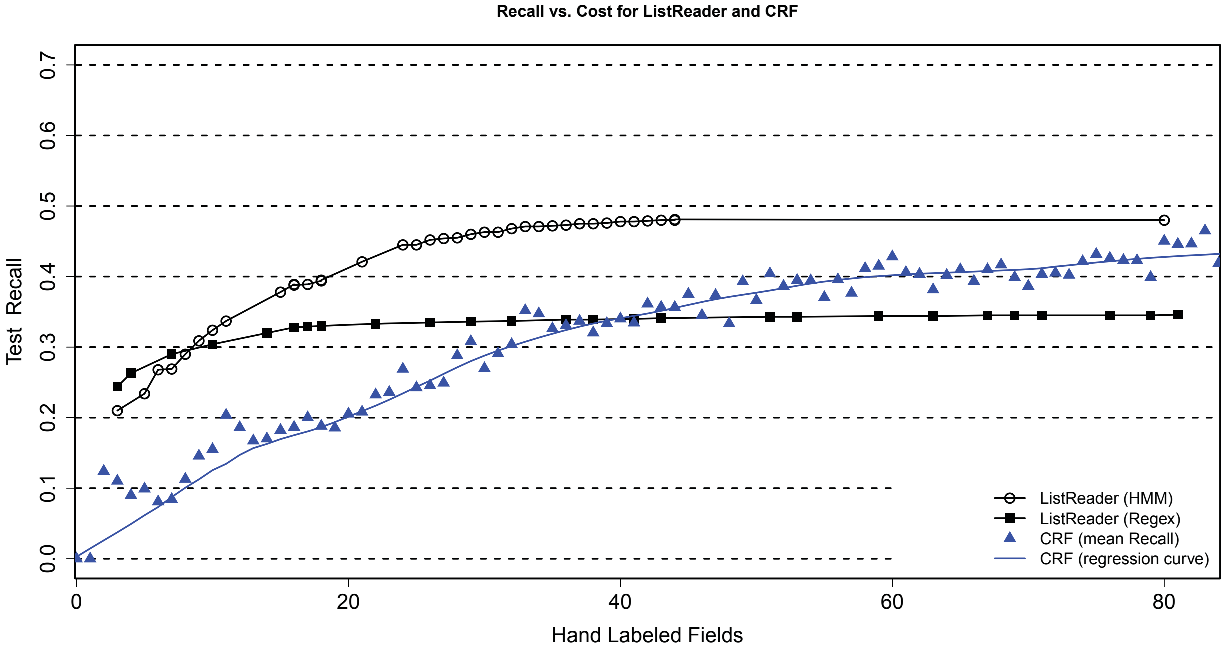


Figure 5.19: Recall Learning Curves for the *Shaver-Dougherty Genealogy*.

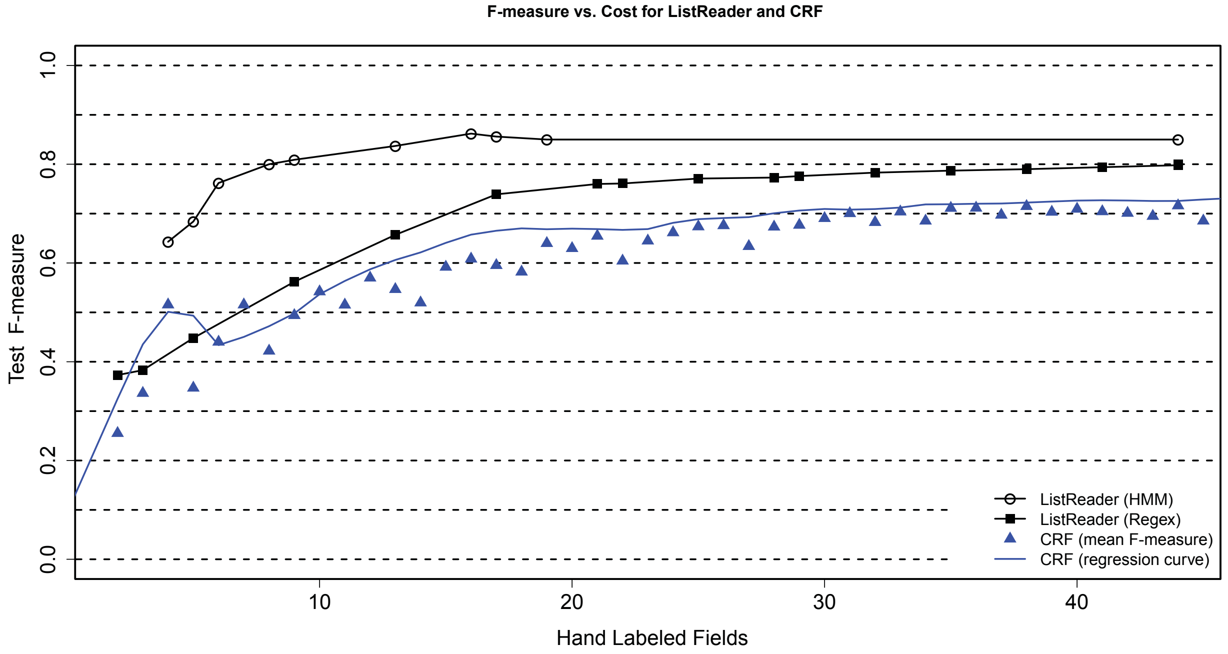


Figure 5.20: F-measure Learning Curves for the *Kilbarchan Parish Register*.

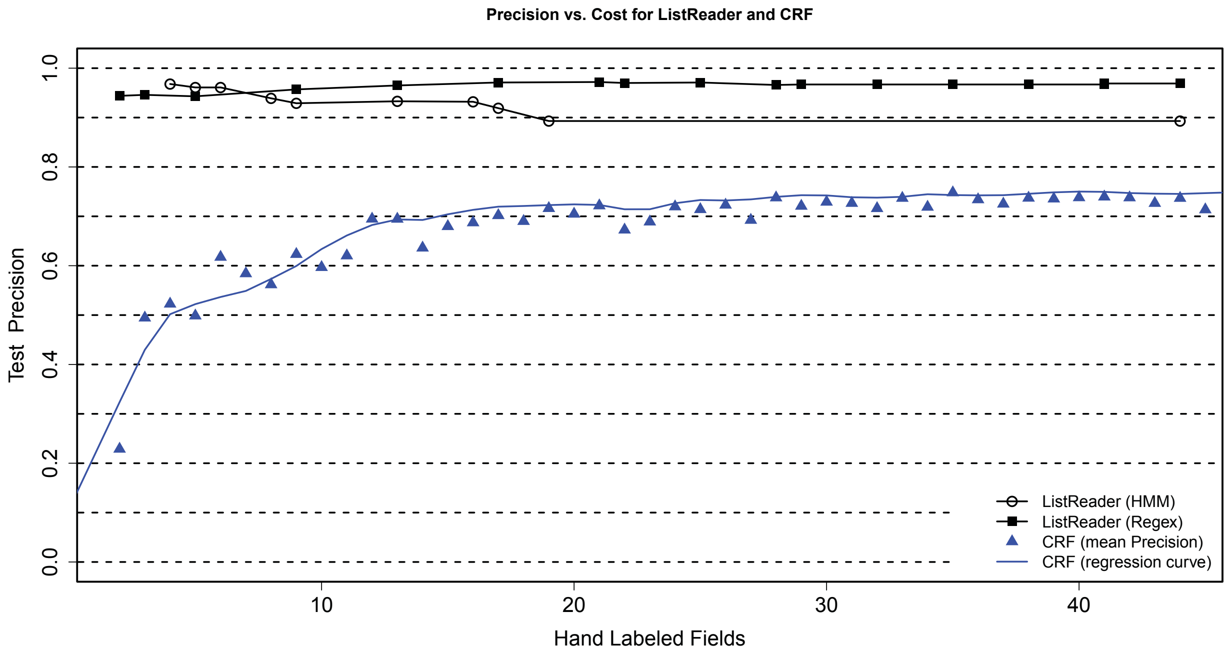


Figure 5.21: Precision Learning Curves for the *Kilbarchan Parish Register*.

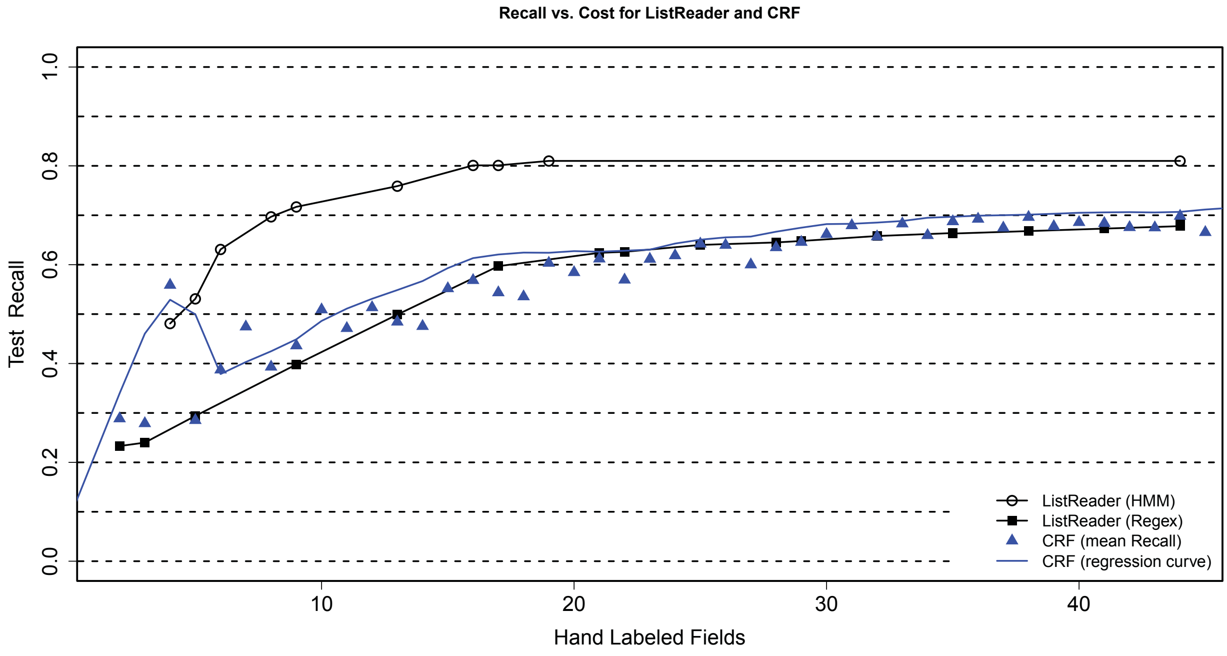


Figure 5.22: Recall Learning Curves for the *Kilbarchan Parish Register*.

The spike in the CRF's recall at Cost = 4 in Figure 5.22 is because the majority of records in the book are child records that contain 4 fields. When the CRF is lucky enough to train on one of these records, it usually does well extracting the other child record information.

Comparing the sizes of the extractors, ListReader (Regex) generated a regular expression that was 319,096 characters long for the *Shaver-Dougherty Genealogy* matching 3,334 records, and one that was 54,600 characters long for the *Kilbarchan Parish Register* matching 268 records. ListReader (HMM) generated an HMM with 2,015 states for the *Shaver-Dougherty Genealogy* matching 3,023 records and an HMM with 255 states for the *Kilbarchan Parish Register* matching 162 records. The HMM matches fewer records than the Regex because it is built from a fraction of the available record parse trees. The key to its improved recall, again, is that each (hand-labeled) HMM record template can match more records than each (hand-labeled) Regex template. Otherwise, the HMM should match less than half of the number of records that the Regex does. The CRF had 353 types of feature values and 28 states for the *Shaver-Dougherty Genealogy* and 191 types of feature values and 15 states for the *Kilbarchan Parish Register*. A smaller number of states probably contributed to its faster running time and lower accuracy compared to the HMM.

Comparing the running time of ListReader, time and space complexity is linear in terms of the size of the input text, but unlike the Regex version, the HMM version is quadratic in the length of the record and the size of the label alphabet. The typical implementation of the training phase of a linear chain CRF is quadratic in both the sizes of the input text and the label set [17], [31]. We ran all extractors on a desktop computer with Java (JDK 1.7), a 2.39 GHz processor, and 3.25 GB of RAM. ListReader (Regex) took 26 seconds to run on the *Kilbarchan Parish Register* and 2 minutes 47 seconds to run on the *Shaver-Dougherty Genealogy*. ListReader (HMM) took 2 minutes 11 seconds to run on the *Kilbarchan Parish Register* and 59 minutes 18 seconds to run on *Shaver-Dougherty Genealogy*. The CRF took 9 seconds on *Kilbarchan Parish Register* and 52 seconds on *Shaver-Dougherty Genealogy*.

5.6 Discussion and Future Work

The errors ListReader (HMM) produces include both precision and recall errors (false positives and false negatives). The most important errors include missing whole records or large segments of records belonging to undiscovered templates. For example, on Page 31 of the *Kilbarchan Parish Register*, ListReader misses the first part of the third record, namely “Cordoner, James, par., and Florence Landiss, par. of Paisley”, because the “par-and” delimiter occurs in only one record cluster and is therefore not recognized as a field group delimiter in our three-page test set. This issue contributes mostly to errors in recall as it causes ListReader to completely miss many fields. It also contributes to a few errors of precision as it causes ListReader to propose a record boundary in the wrong place (just past the missing information).

ListReader (HMM), as compared to ListReader (Regex), does relatively well in recall for the same reason it does relatively poorly in precision—by matching more text. By design, it uses only one “feature” per word token, and that feature is easily derived from the text, itself, without large knowledge resources. This is in contrast to our implementation of the CRF which, instead of removing information as our HMM does, the feature extractors add information. This makes the comparison CRF a less scalable option in terms of development cost over multiple domains or text genres compared to our HMM whose main operating principle could be stated as “carefully throwing out the right kind of information.” The technique of using semantic or lexical resources is somewhat more complicated in our work because of OCR errors that make dictionary matching more difficult. We thus do not currently rely on them.

On the other hand, adding semantic constraints to the HMM would likely help prevent some of its precision errors, such as labeling an “m.” as a surname at the beginning of a line whose other text matched a known record pattern. Future work should investigate adding such semantic features or constraints to ListReader in a way that is cost-effective, for example using self-training, co-training, or bootstrapping that learns semantic categories from the input text, itself. We could also train ListReader from examples labeled automatically by other extractors, from wrappers trained on other books, or from examples that match a database of known facts such as the work in [20], with the added costs associated with those resources. Since the final mapping from HMM states to labels and predicates is the only step currently needing human labeled examples, adding a technique that utilizes automatically-labeled examples would make our approach completely unsupervised and scalable in terms of supervision cost.

Looking further ahead to applying ListReader to arbitrary lists, we should consider those that are less structured and more like natural language. The current implementation assumes that any given field group delimiter (and, in turn, every connected field group) has a fixed semantics regardless of where in a record it occurs, and regardless of whether that field group delimiter may appear in more than one location. For example, we see a few records in the *Kilbarchan Parish Register* of the following form "`\nMarshall, William, in Lochermilne, and Jean Reid, in Killallan\n`". ListReader will ask for a label from the user for only one of the "`in`" field groups and therefore will label them both as the same, e.g. both associated with the husband or both associated with the wife, but not each associated with the correct spouse. Future versions of ListReader should overcome this limitation while preserving the labeling efficiency, for example by learning that the "`and`" delimiter separates field groups associated with the husband and the wife which should allow ListReader to then distinguish between the two "`in`" field groups during parsing, labeling, and mapping.

5.7 Related Work

Having described and evaluated ListReader, we now compare it with related research—wrapper induction in support of information extraction from semi-structured documents (Subsection 5.7.1) and unsupervised learning for extraction models (Subsection 5.7.2).

5.7.1 Automated Information Extraction from Lists

In general, related projects in the web wrapper induction literature [16] are almost universally applied to clean text—mostly to structured HMTL documents, and sometimes to semi-structured lists—allowing them to learn record patterns from as little as one page of input. The information extraction task of these systems is similar to ListReader's in that they extract, label, and group fields together that belong to the same record, and there can be multiple isolated records on a page. On the other hand, unlike ListReader, many wrapper induction approaches look for contiguous records, and those approaches that, like ListReader, are based on limited user input do not work with plain text (non-HTML) input and do not recognize multiple orderings of extracted fields. Those that work with plain text, like ListReader, use syntactic and semantic features such as part of speech tags and WordNet categories, which ListReader does not need to use.

Choices in wrapper formalism include sets of left and right field context expressions ([41], [5]), xpaths ([20]), finite state automata ([43]), and conditional random fields ([22], [32]). These formalisms generally rely on consistent landmarks that are not available in OCRed lists for three reasons: OCRed list text is less consistently structured than machine-generated HTML pages, OCRed text does not contain HTML tags, and field delimiters and content in OCRed documents often contain OCR and typographical errors. Furthermore, none of these projects address all of the steps necessary to complete the process of the current research such as list finding, record segmentation, and field extraction.

The wrapper induction work most closely related to ListReader is IEPAD [15]. IEPAD consists of a pipeline of four steps: token encoding, PAT tree construction, pattern filtering, and rule composing. Like ListReader, IEPAD must deal with a trade-off between coarsely encoding the text to reduce the noise enough to find patterns and finely encoding the text to maintain all the distinctions specified by the output schema. Also, PAT trees are related to suffix trees and share similar time and space properties. However, we note some important differences. The algorithmic complexity of IEPAD's wrapper construction phase appears to be quadratic because of its reliance on multiple string alignment, while ListReader's is linear. ListReader must use a different means of encoding (conflating) text than IEPAD so it can preserve more fine grained structure. IEPAD apparently cannot extract fields that are not explicitly delimited by some kind of HTML tag and looks only for contiguous records. Also, it appears that IEPAD users must identify pages containing target information; a ListReader user does not need to do so. IEPAD requires users to select patterns because the system may produce more than one pattern for a given type of record. ListReader automatically selects patterns among a set of alternatives. IEPAD users must also provide labels for each pattern, which is similar to the work of ListReader users, but is likely more difficult because it forces users to interpret induced patterns rather than original text, which raises the required user skill level. ListReader also minimizes the amount of supervision needed to extract a large volume of data by integrating an interactive labeling process into grammar induction, something IEPAD does not do. Lastly, ListReader reduces the cost of extracting information because of its unique combination of global pattern detection and active sampling—it focuses a user's effort on the most common patterns first, a valuable property not explored in any related information-extraction research.

5.7.2 Unsupervised Learning for Extraction Models

There are many wrapper induction projects applied to web pages that have a strong element of unsupervised machine learning, such as [41], [5], [20], and [43]. These and other related research projects do not solve our targeted problem. Most do not address lists, specifically, and none address plain OCRed text. As Gupta and Sarawagi say ([32]), the vast majority of methods of extraction of records from unstructured lists assume the presence of labeled unstructured records for training and a few assume a large database of structured records. None of these projects address all of the steps necessary to complete the process of the current research such as list finding, record segmentation, field extraction, and mapping to an expressive ontology.

A common and mathematically motivated means of unsupervised HMM induction is the Baum-Welch algorithm, an instance of the iterative Expectation-Maximization algorithm (EM). Baum-Welch finds the MLE parameters of an HMM in either unsupervised or semi-supervised learning scenarios. In either case, text without manually-provided labels are assigned those labels that are most probable given the current HMM parameters, and those HMM parameters are in turn set from the most probable label distributions given the parameters set on the previous iteration. Grenager et al. ([28]) use EM to train an HMM in both unsupervised and semi-supervised scenarios to extract fields from plain text records, including bibliographic citations and classified advertisements.

They supplement EM with a few domain-dependent biases to prefer diagonal (self) transitions and recognize boundary tokens (punctuations). They report that the accuracy of the unsupervised approach starts low but is improved with the added biases. Furthermore, before adding the biases, their semi-supervised approach performed worse than supervised learning given the same number of hand-labeled examples, according to our reproduction of their work. The fields they extract are coarse-grained, such that a sequence of author names in a bibliographic citation is considered one homogeneous segment. Our work differs from theirs in that we set the HMM parameters from record structure proposed by a separate phrase grammar that we induce automatically and separately (without any connection to the HMM). We also extract more fine-grained information, e.g. individual person names and parts of those names, to improve the richness of the resulting data.[6] Therefore, their self-transition bias would not be appropriate in our work. Also, Grenager et al. assume that list records have been found and extracted before their process begins, which we do not assume for ours. Unlike the semi-supervised part of their work, we do not perform any training of the HMM's structure or parameters using hand labeled data which may be a more scalable approach given a large input corpus.

Elmeleegy et al. ([22]) present an algorithm to automatically convert a source HTML list into a table, with no hand-labeled training data and no output labeling of fields or columns. They segment fields in records automatically using the following sources of information to predict which words should be split and which should remain together: (1) sets of "data type" regular expressions including common numeric entity patterns, (2) an n-gram language model producing internal cohesiveness and external in-cohesiveness scores, and (3) a thresholded count of the number of cells matched in a corpus of extracted table cells. They combine these sources of evidence using a weighted average. They also correct errors in the first pass of segmentation by counting fields, forcing all records to be segmented into no more than the most common number of fields, and aligning shorter records using a modification of the Needleman-Wunsch algorithm. Like Grenager

[6] The benefits of a fine-grained ontology include the following: (1) it can allow an ontology user to evolve the schema without either retraining the extraction model or manually restructuring individual fields within the resulting database and (2) it can improve the accuracy and versatility of downstream processes such as querying, record linkage, and ontology mapping.

et al., they perform field segmentation and alignment but do not appear to perform list discovery or record segmentation as we do. They also do not label fields or fully extract information, and they target HTML lists which may contain additional formatting clues not present in our OCR text. Unlike us, they assume that the order of fields does not change between list entries. Unsupervised techniques like theirs target web-scale applications and they also rely on a web-scale corpus. Therefore, they avoid hand-labeling of training data. Their source table data is a massive collection of tables from the web. Using massive amounts of web data is a common technique among recent web wrapper research that rely on the sheer size of the web as a key resource for their system. We do not use web-scale data resources. They assume there are not many optional fields in their input data which is not true of our data. Forcing the number of fields/columns to equal the mode of the field count per row discovered in the first pass will not work correctly for many lists because there can be optional fields which do not often occur.

Gupta and Sarawagi ([32]) convert HTML source lists on the web into tables that match and augment an incomplete user-provided table. Their unsupervised approach first ranks lists with a Lucene query, based on the words in the user-provided table. Second, they label candidate fields in the source list records as training data by marking text in the list records that match text in the columns of the user-provided table. Third, they train a separate CRF for each source list using the automatically labeled records of the list and then apply the CRF to the rest of the records of that list. This effectively produces tables from the lists. They finally merge and rank the rows of the resulting tables and returns the top ranked rows of the final table to the user. Rows that repeat often in source lists and which are given high confidence scores by the CRF are ranked high. This work is similar to ours in that they train a statistical sequence model on the text of lists labeled by a separate, automatic process. It differs from ours in that their source text (web pages) have no OCR errors and have more structure making it easier to find lists, segment records, and identify fields. They do not need to complete a mapping from text fields to ontology predicates, they only need to align user-provided fields with fields in a list record. They do not seem to (or need to) segment records in lists before extracting fields. They have a much larger source of potential lists than we do and only need to find some with high accuracy, not all of them. In our project, we evaluate against an ideal of extracting all list records from a book. This work, as well as the other two, do not extract richly- and explicitly-structured data suitable for ontology population as we do.

5.8 Concluding Remarks

ListReader addresses the problem of extracting information from OCRed lists for ontology population. It requires little effort to apply to a new book, is specialized to recognize and model list structures, and is tolerant of OCR errors. Our HMM implementation of ListReader demonstrates a novel way to set the structure and parameters of an HMM automatically for the task of populating an expressive conceptual model with information from lists in OCRed text. It also demonstrates a way to minimize the work necessary for completing the HMM wrapper by manually associating automatically-selected HMM states with ontology predicates. ListReader performs well in terms of accuracy, user labeling cost, time and space complexity, and required knowledge engineering—outperforming the comparison systems in terms of most criteria including the most important measure: accuracy achieved relative to minimal manual annotation cost.

Chapter 6

Conclusions

6.1 Concluding Remarks and Lessons Learned

Our goal was a complete, end-to-end solution to the problem of populating a rich ontology from the records in semi-structured lists found in OCRed text. Further, our aim in finding a solution was to balance multiple criteria: accuracy, computational complexity, application cost, generality, and completeness. These criteria compete with each other, and optimizing one will naturally lead to doing poorly in another. Though it is most likely impossible to do perfectly over all criteria simultaneously, we think it is a mistake to ignore any of them. Sometimes we may find convenient compromises that do reasonably well in multiple criteria. For example and counter-intuitively, to deal with the low-cost, high recall, and OCR criteria, we discovered a good compromise: instead of adding information as most work in feature extraction for machine learning research does, we found that it is better to lose information—carefully throwing out just the right information. Like squinting to find patterns, we threw out the identity of most words. This is easy and cheap to do. When price is not an issue, adding information may yet be a good idea. But we found in this project that when price is an issue in conjunction with accuracy, it is useful to eliminate detailed information and look at the text at a higher-level of abstraction. Certain aspects of the research converged toward this common technique: OCR errors make dictionary matching harder, the low-cost requirement made creating dictionaries less desirable, the low time and space complexity requirement motivated processing less information, and the rich ontology requirement is consistent with extracting information from complete records. We ended up using a global or holistic approach instead of a local approach, in multiple ways. We align text throughout the book instead of trying to align neighboring records one at a time. We conflate text to lose superfluous details and look at whole-record patterns instead of individual tokens or other detailed features of tokens. Records as a whole contain useful, invariant structure. Not worrying about the details of word features was helpful; OCR errors and natural variations among fields will make it hard to extract features from individual words anyway.

The Regex and HMM approaches have different strengths and weaknesses. Most notable and consistent is that the Regex approaches have higher precision and lower recall while the HMM approaches have a more balanced precision and recall and therefore higher F-measure. Either profile could be beneficial, depending on the application. Here we discuss just a few. High precision would benefit a recommender system (also called a recommendation engine). Consider a social networking or family history website in which users build a social network or a family tree. Recommender systems are becoming common features of such websites. A recommender system will propose a record of a person to add to the user's social network or tree. On one hand, the user is not expecting a predetermined number of recommendations. On the other hand, the website does not want to lose credibility by wasting the time of its paying customers with many false positives. In this situation, the extraction of highly precise (and rich) information about a potentially-related person is important for making precise recommendations.

On the other hand, higher recall and F-measure would benefit a targeted search for a rare item. Consider a scenario within genealogy research in which it is difficult to find any information about a certain important ancestor. In this case, the genealogy website would want to offer the user a search tool that has higher recall without sacrificing precision too much, otherwise the user may not be able to find any information about the obscure ancestor or may have too many candidate records to search through manually.

We see a tendency in research to focus on improving accuracy while ignoring the hidden increases in costs associated with those improvements [11]. Well-developed statistical models such as the CRF can perform well on a number of tasks, but that performance comes with additional costs in terms of knowledge and feature engineering, manual labeling of training data, and other domain-, genre-, and task-specific refinements. We conclude that a simple pipeline architecture using a global approach to pattern recognition can detect and extract richly structured information from lists in OCRed text in a scalable, portable, and effective manner.

6.2 Limitations and Future Work

Table 1.1 summarizes the four papers in Chapters 2–5 and shows that, by the fourth paper, we have achieved "good" results over all nine of our targeted success criteria using the global HMM induction strategy. Our previous approach, global regex induction (Chapter 4), achieved nearly the same level in all criteria, except for a less balanced trade-off in four areas. It does well in algorithm time and space complexity and in precision, but does less well in recall and in OCR and noise tolerance. We believe these are related in that increasing recall with further work will likely decrease precision and will likely require more tolerance of noise. On the other hand, we believe the algorithmic complexity can most likely remain linear. The key strengths of ListReader (Global Regex) is a combination of speed and precision which are likely a valuable combination in some applications despite the lower recall. Future research should investigate ways of adding the strength of the HMM, which is higher in recall, to the Regex without sacrificing precision and speed. One approach might be to add to the Regex the HMM's ability to cover its entire learning curve quickly, e.g., by sharing the labels a user provides for common patterns with similar but less-common patterns.

In analyzing the errors of our final approach, ListReader (Global HMM), we see that it produced both precision and recall errors (false positives and false negatives). The HMM does relatively well in recall for the same reason it does relatively poorly in precision—by matching more text. By design, it uses only one "feature" per word token, and that feature is easily derived from the text, itself, without additional knowledge resources. This is in contrast to the CRF which, instead of removing information as our HMM does, the feature extractors add information. This makes the CRF a less scalable option in terms of development cost over multiple domains or text genres compared to our HMM whose main operating principle could be stated as "carefully throwing out just the right information". On the other hand, adding semantic constraints to the HMM would likely help prevent some of its precision errors, such as labeling an "m." as a surname at the beginning of a line that otherwise matched a known record pattern. Future work should investigate adding such semantic features or constraints to ListReader in a way that is cost-effective, for example using self-supervision or a bootstrapping mechanism that learns its own semantic word clusters from the input text, itself. We could also train ListReader from examples labeled automatically by other extractors, from wrappers trained on other books, or from examples that match a database of known facts, with the added costs associated with those resources. The technique of using external semantic or lexical resources is complicated in our work by the common OCR errors that make dictionary matching more difficult. Jointly correcting OCR errors and discovering semantic categories is an open area of research that would be interesting to explore within the ListReader framework.

It may be possible to use a CRF as a wrapper instead of a Regex or HMM and make use of its more expressive feature set. The expressive set of features the CRF uses allows it to continue learning more from more training data, as we can see from the recall learning curves in Chapter 5. The CRF tends to continue learning after ListReader has hit a plateau. One way to implement this would be to train the CRF on the high-precision labeling of ListReader (Global Regex). Another way would be to construct the CRF from parse trees in the way we construct the HMM in Chapter 5. On the other hand, using the CRF and its larger set of word features would depart from one of our evaluation criteria: little or no feature engineering.

We also have considered trying to combine the higher precision of ListReader (Regex) with the higher recall of ListReader (HMM) in an ensemble of the two. We manually computed the precision, recall, and F-measure of what an ensemble would produce with the following logic: output labels for all the records discovered by the Regex, then output the labels for all the records that the HMM outputs that the Regex missed. This should preserve everything the Regex produced with high precision, while adding to that some of the records that allowed the HMM to reach higher recall. Compared to the Regex as a baseline, recall would go up by over 10%, but precision would go down by about 5%. F-measure would also go up by about 5%. These evaluation metrics appear similar to the HMM's, but are sometimes worse. Considering this and the added challenge of ensuring that the cost of manually providing labels for both extractors is as low as providing labels for just one extractor, we are uncertain that such an ensemble could outperform the HMM. The insight this exercise gives us is that many or most of the false positives of the HMM are found in the records it extracts that the regex does not, i.e. the less-than-typical record structures. This also means that, though we could increase the recall of the Regex by making it as lenient as the HMM, and we could increase the precision of the HMM by making it as selective as the Regex, little if any progress will be made without more fundamental improvements in the unsupervised grammar induction that both depend on, in such a way that the unusual structures in certain uncommon records are more correctly modeled. Sometimes records or parts of records are too uncommon for ListReader to recognize the record as a record and sometimes field groups are uncommon enough that ListReader does not correctly recognize the field group templates and in turn does not connect fields in a common record structure that was labeled by a user with the same fields in an uncommon record structure further down the long tail of the learning curve. These are instances where the HMM makes more guesses than the Regex because it is more lenient to structural changes. Future work should therefore focus on correctly modeling records with unusual structure.

Long records are a common source of both precision and recall errors among all versions of ListReader. Longer records span multiple lines, and ListReader often continues to interpret a single line record fragment as a full record. Giving the user a way to correct mistakes of this kind will likely improve both precision and recall significantly. This added interaction would necessitate either having ListReader change the structure of the wrapper after the initial unsupervised phase or else perform true active learning during the initial grammar induction phase to guide the initial discovery of record clusters and wrapper templates, most naturally in the record selection steps.

ListReader (Global HMM) is currently limited in how it utilizes user feedback. For example, the generic insertion states added during HMM construction may match many textual patterns that will not necessarily all have the same final labels. Future work should change Global ListReader so it creates a new sub-HMM for each pattern matched by a each insertion state as we did with the Local HMM in Chapter 3.

Another source of error for the two global approaches is the input parameters and conflation rules. Our evaluation metrics would likely improve if both kinds of inputs could be tuned to each book. The user is free to develop additional conflation rules and order them as seems best for each book. These could include rules based on regular expressions (e.g., common date formats) or dictionaries (e.g., place names) or any other pattern the user finds in the input text that should be treated as invariant across records. We did not apply this strategy in our research (except within the comparison CRF's feature extractors) in order to convincingly demonstrate a low-cost solution. On the other hand, another research angle could be to allow the user to develop document-specific conflation rules to be leveraged within the ListReader framework, and then show that the manual work in constructing these rules is still lower than having someone write complete regular expressions from scratch, while also achieving the high accuracy that a fully customized document grammar would produce.

Our application of MDL, if successful, could have allowed ListReader to dynamically create conflation rules during grammar induction and select the right rules, and the right over-all level of abstraction, for each book. We believe that a compromise between that approach and the final ListReader will likely work better than either approach. In such an approach, we would develop a larger set of alternative conflation rules up front, covering a wide range of genres and document types, and use a hypothesis testing technique to select the right ones for each book.

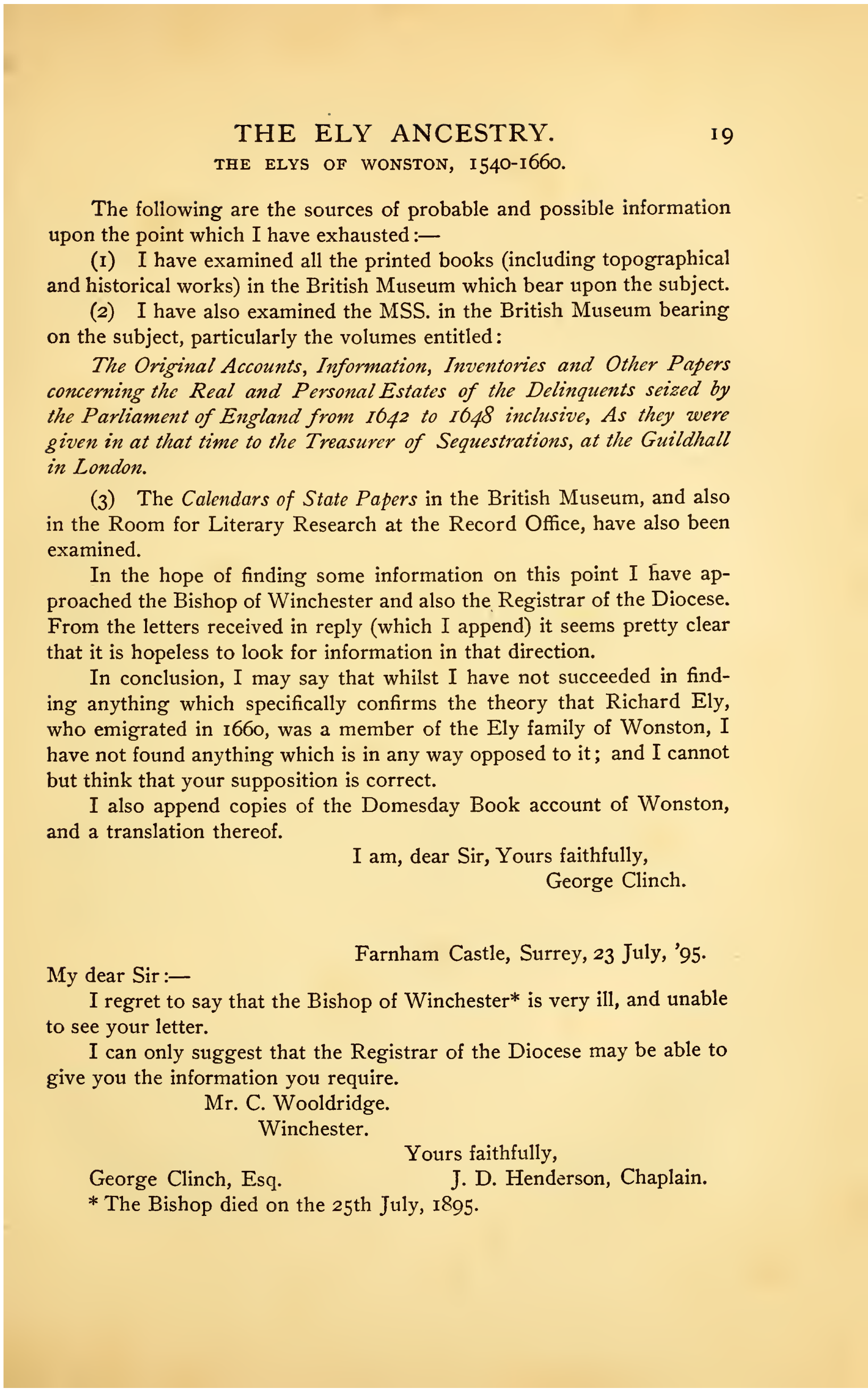
To expand the usefulness of the extracted information for certain applications like family history research, future work should investigate ways of recognizing relationships that connect labeled fields across distinct records. For example, consider the parent-child relationship between each child in the second-level records of a nested lists in a family history book and the parents named in the first-level record immediately before them. We expect that a larger, hierarchical or recursive wrapper structure may be necessary to model these kinds of nested lists.

Despite these limitations, we have demonstrated a simple, scalable, and effective way of simultaneously improving the accuracy and decreasing the cost of extracting information from OCRed lists. ListReader effectively combines unsupervised grammar induction with active sampling and knowledge mapping to identify, extract, and structure data in lists of noisy OCRed text for effective ontology population. ListReader performs well in terms of accuracy, user labeling cost, time and space complexity, and required knowledge engineering—outperforming the comparison systems in terms of F-measure as a function of labeling cost with statistical significance.

Appendix A

Example Prose, List, and Index Pages from Ely and Shaver

Family history books generally include three kinds of text: (1) Prose/narrative, (2) family lists, and (3) name-index lists (at the back of the book). We have taken examples of these three main types of pages from both of the books discussed in this paper: *The Ely Ancestry* and *Shaver-Dougherty Genealogy*. The examples appear below in Figures A.1, A.3, A.5, A.7, A.9, A.11. The OCR text of these pages appear in Figures A.2, A.4, A.6, A.8, A.10, A.12.



THE ELY ANCESTRY. 19

THE ELYS OF WONSTON, 1540-1660.

The following are the sources of probable and possible information upon the point which I have exhausted:—

(1) I have examined all the printed books (including topographical and historical works) in the British Museum which bear upon the subject.

(2) I have also examined the MSS. in the British Museum bearing on the subject, particularly the volumes entitled:

The Original Accounts, Information, Inventories and Other Papers concerning the Real and Personal Estates of the Delinquents seized by the Parliament of England from 1642 to 1648 inclusive, As they were given in at that time to the Treasurer of Sequestrations, at the Guildhall in London.

(3) The *Calendars of State Papers* in the British Museum, and also in the Room for Literary Research at the Record Office, have also been examined.

In the hope of finding some information on this point I have approached the Bishop of Winchester and also the Registrar of the Diocese. From the letters received in reply (which I append) it seems pretty clear that it is hopeless to look for information in that direction.

In conclusion, I may say that whilst I have not succeeded in finding anything which specifically confirms the theory that Richard Ely, who emigrated in 1660, was a member of the Ely family of Wonston, I have not found anything which is in any way opposed to it; and I cannot but think that your supposition is correct.

I also append copies of the Domesday Book account of Wonston, and a translation thereof.

I am, dear Sir, Yours faithfully,
George Clinch.

Farnham Castle, Surrey, 23 July, '95.

My dear Sir:—

I regret to say that the Bishop of Winchester* is very ill, and unable to see your letter.

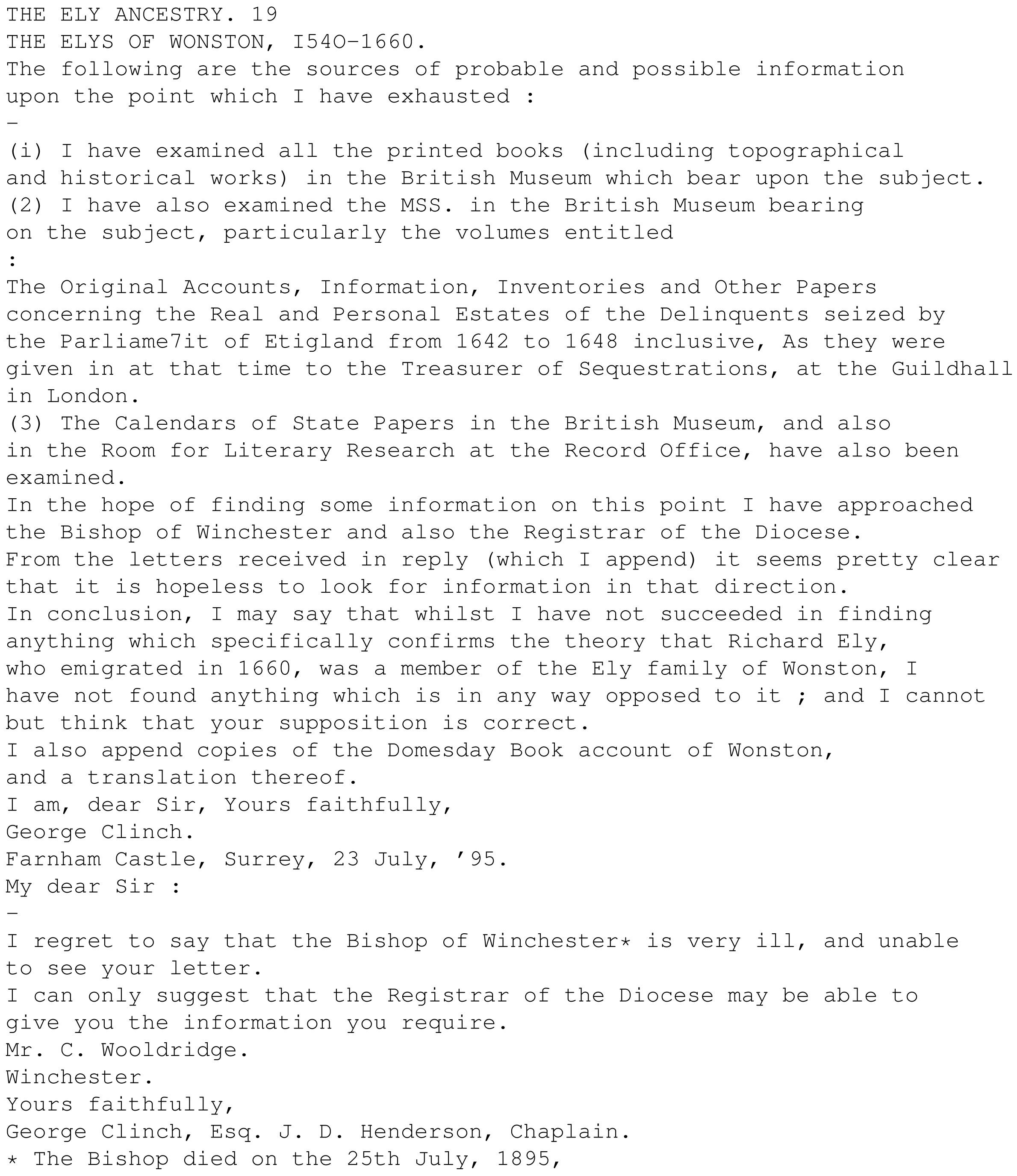
I can only suggest that the Registrar of the Diocese may be able to give you the information you require.

Mr. C. Wooldridge.
Winchester.

Yours faithfully,
George Clinch, Esq. J. D. Henderson, Chaplain.

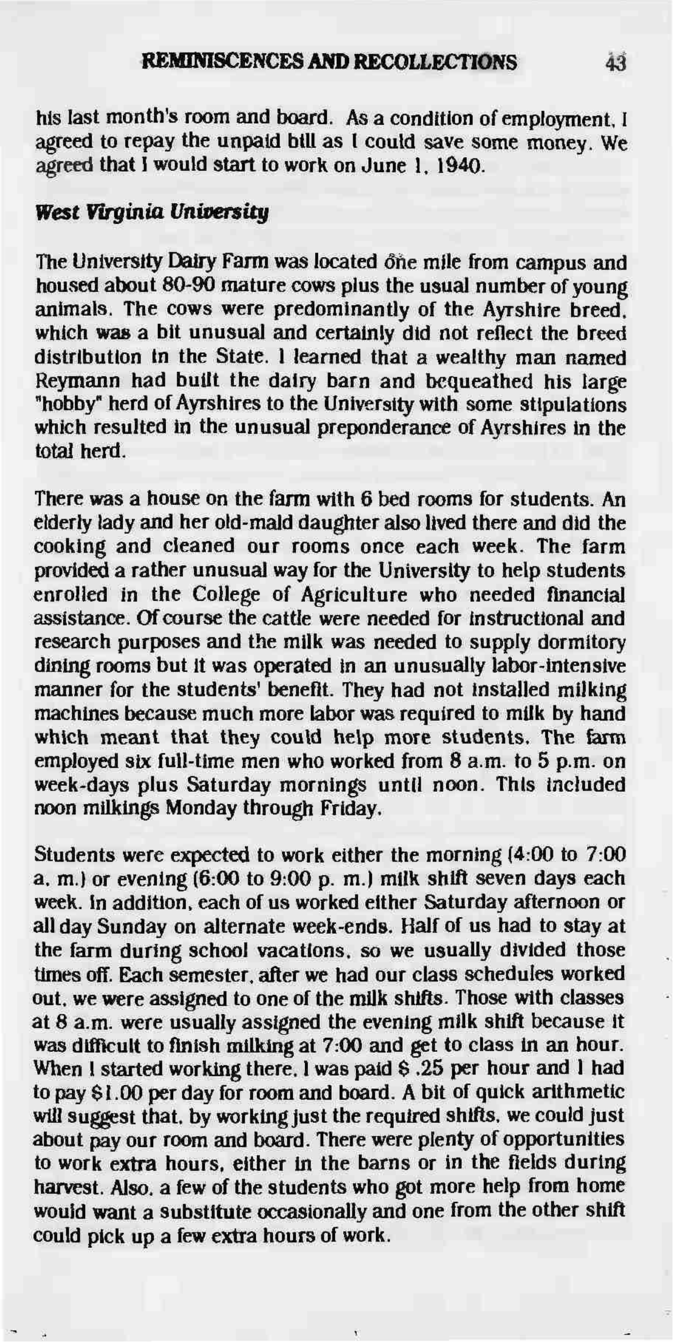
* The Bishop died on the 25th July, 1895.

Figure A.1: Prose Page in *The Ely Ancestry*, Page 19.



```
THE ELY ANCESTRY. 19
THE ELYS OF WONSTON, I54O-1660.
The following are the sources of probable and possible information
upon the point which I have exhausted :
-
(i) I have examined all the printed books (including topographical
and historical works) in the British Museum which bear upon the subject.
(2) I have also examined the MSS. in the British Museum bearing
on the subject, particularly the volumes entitled
:
The Original Accounts, Information, Inventories and Other Papers
concerning the Real and Personal Estates of the Delinquents seized by
the Parliame7it of Etigland from 1642 to 1648 inclusive, As they were
given in at that time to the Treasurer of Sequestrations, at the Guildhall
in London.
(3) The Calendars of State Papers in the British Museum, and also
in the Room for Literary Research at the Record Office, have also been
examined.
In the hope of finding some information on this point I have approached
the Bishop of Winchester and also the Registrar of the Diocese.
From the letters received in reply (which I append) it seems pretty clear
that it is hopeless to look for information in that direction.
In conclusion, I may say that whilst I have not succeeded in finding
anything which specifically confirms the theory that Richard Ely,
who emigrated in 1660, was a member of the Ely family of Wonston, I
have not found anything which is in any way opposed to it ; and I cannot
but think that your supposition is correct.
I also append copies of the Domesday Book account of Wonston,
and a translation thereof.
I am, dear Sir, Yours faithfully,
George Clinch.
Farnham Castle, Surrey, 23 July, '95.
My dear Sir :
-
I regret to say that the Bishop of Winchester* is very ill, and unable
to see your letter.
I can only suggest that the Registrar of the Diocese may be able to
give you the information you require.
Mr. C. Wooldridge.
Winchester.
Yours faithfully,
George Clinch, Esq. J. D. Henderson, Chaplain.
* The Bishop died on the 25th July, 1895,
```

Figure A.2: Prose Page (OCR Text) in *The Ely Ancestry*, Page 19.



REMINISCENCES AND RECOLLECTIONS 43

his last month's room and board. As a condition of employment, I agreed to repay the unpaid bill as I could save some money. We agreed that I would start to work on June 1, 1940.

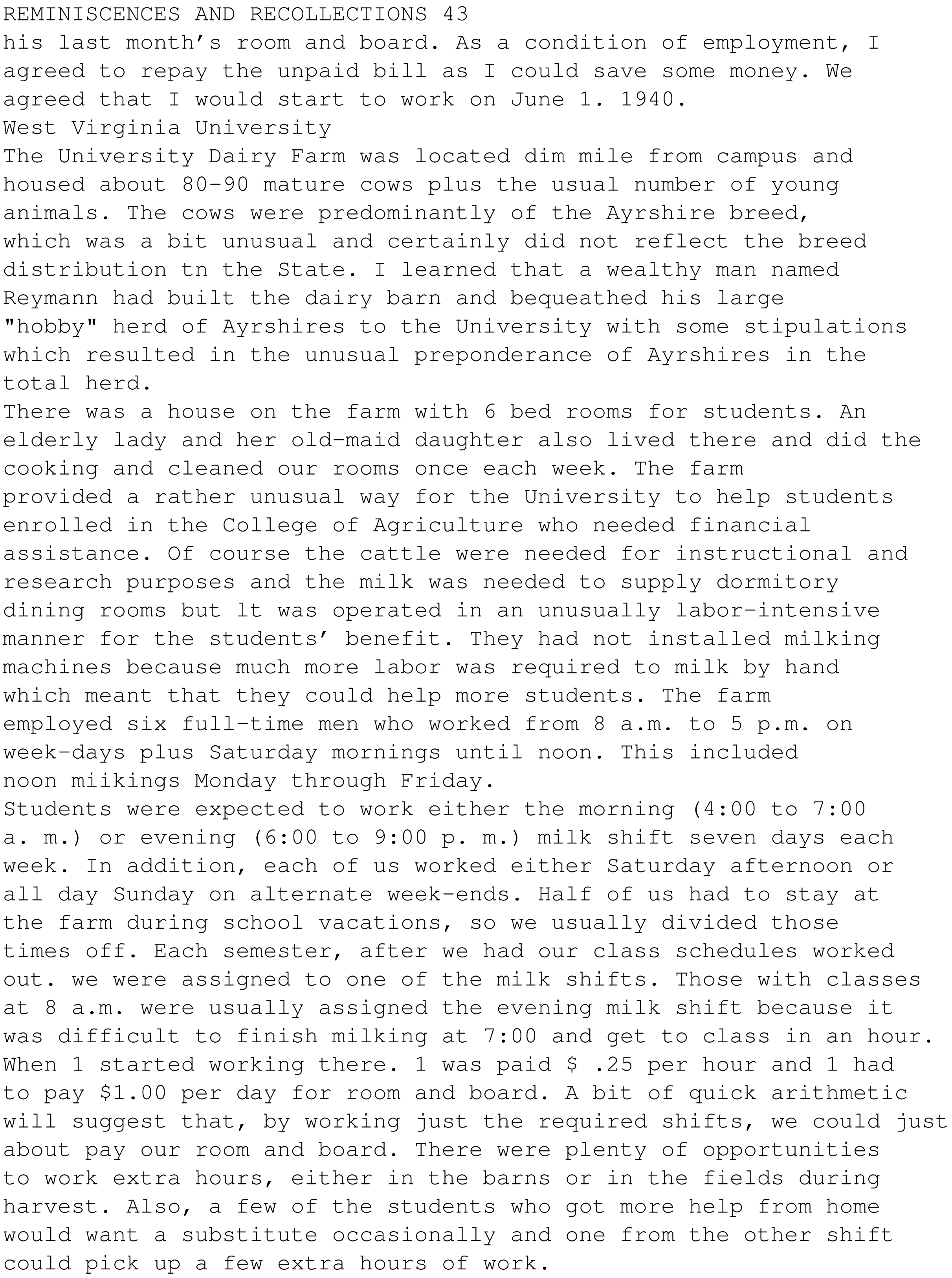
West Virginia University

The University Dairy Farm was located one mile from campus and housed about 80-90 mature cows plus the usual number of young animals. The cows were predominantly of the Ayrshire breed, which was a bit unusual and certainly did not reflect the breed distribution in the State. I learned that a wealthy man named Reymann had built the dairy barn and bequeathed his large "hobby" herd of Ayrshires to the University with some stipulations which resulted in the unusual preponderance of Ayrshires in the total herd.

There was a house on the farm with 6 bed rooms for students. An elderly lady and her old-maid daughter also lived there and did the cooking and cleaned our rooms once each week. The farm provided a rather unusual way for the University to help students enrolled in the College of Agriculture who needed financial assistance. Of course the cattle were needed for instructional and research purposes and the milk was needed to supply dormitory dining rooms but it was operated in an unusually labor-intensive manner for the students' benefit. They had not installed milking machines because much more labor was required to milk by hand which meant that they could help more students. The farm employed six full-time men who worked from 8 a.m. to 5 p.m. on week-days plus Saturday mornings until noon. This included noon milkings Monday through Friday.

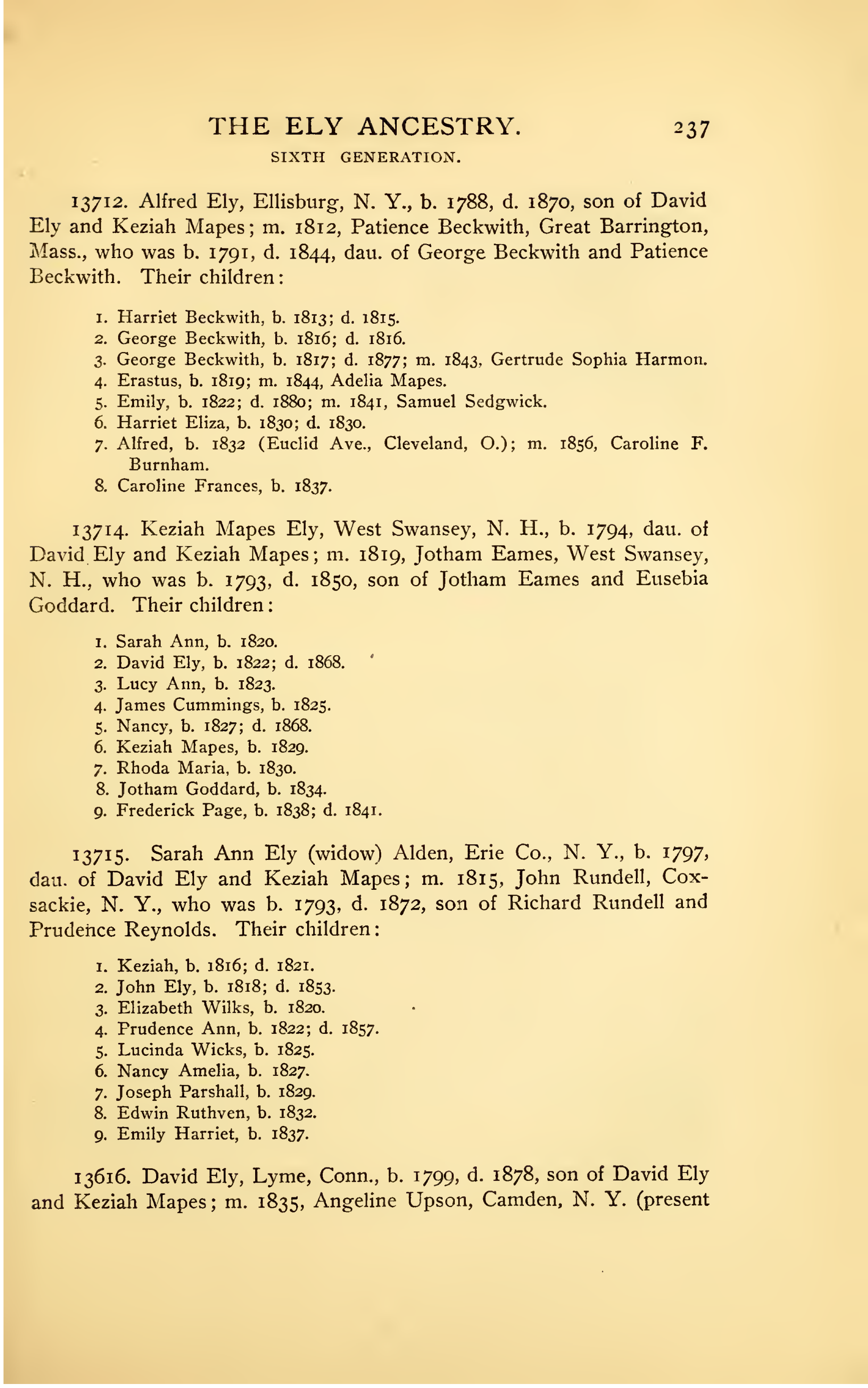
Students were expected to work either the morning (4:00 to 7:00 a. m.) or evening (6:00 to 9:00 p. m.) milk shift seven days each week. In addition, each of us worked either Saturday afternoon or all day Sunday on alternate week-ends. Half of us had to stay at the farm during school vacations, so we usually divided those times off. Each semester, after we had our class schedules worked out, we were assigned to one of the milk shifts. Those with classes at 8 a.m. were usually assigned the evening milk shift because it was difficult to finish milking at 7:00 and get to class in an hour. When I started working there, I was paid $ .25 per hour and I had to pay $1.00 per day for room and board. A bit of quick arithmetic will suggest that, by working just the required shifts, we could just about pay our room and board. There were plenty of opportunities to work extra hours, either in the barns or in the fields during harvest. Also, a few of the students who got more help from home would want a substitute occasionally and one from the other shift could pick up a few extra hours of work.

Figure A.3: Prose Page in *Shaver-Dougherty Genealogy*, Page 43.



```
REMINISCENCES AND RECOLLECTIONS 43
his last month’s room and board. As a condition of employment, I
agreed to repay the unpaid bill as I could save some money. We
agreed that I would start to work on June 1. 1940.
West Virginia University
The University Dairy Farm was located dim mile from campus and
housed about 80-90 mature cows plus the usual number of young
animals. The cows were predominantly of the Ayrshire breed,
which was a bit unusual and certainly did not reflect the breed
distribution tn the State. I learned that a wealthy man named
Reymann had built the dairy barn and bequeathed his large
"hobby" herd of Ayrshires to the University with some stipulations
which resulted in the unusual preponderance of Ayrshires in the
total herd.
There was a house on the farm with 6 bed rooms for students. An
elderly lady and her old-maid daughter also lived there and did the
cooking and cleaned our rooms once each week. The farm
provided a rather unusual way for the University to help students
enrolled in the College of Agriculture who needed financial
assistance. Of course the cattle were needed for instructional and
research purposes and the milk was needed to supply dormitory
dining rooms but lt was operated in an unusually labor-intensive
manner for the students’ benefit. They had not installed milking
machines because much more labor was required to milk by hand
which meant that they could help more students. The farm
employed six full-time men who worked from 8 a.m. to 5 p.m. on
week-days plus Saturday mornings until noon. This included
noon miikings Monday through Friday.
Students were expected to work either the morning (4:00 to 7:00
a. m.) or evening (6:00 to 9:00 p. m.) milk shift seven days each
week. In addition, each of us worked either Saturday afternoon or
all day Sunday on alternate week-ends. Half of us had to stay at
the farm during school vacations, so we usually divided those
times off. Each semester, after we had our class schedules worked
out. we were assigned to one of the milk shifts. Those with classes
at 8 a.m. were usually assigned the evening milk shift because it
was difficult to finish milking at 7:00 and get to class in an hour.
When 1 started working there. 1 was paid $ .25 per hour and 1 had
to pay $1.00 per day for room and board. A bit of quick arithmetic
will suggest that, by working just the required shifts, we could just
about pay our room and board. There were plenty of opportunities
to work extra hours, either in the barns or in the fields during
harvest. Also, a few of the students who got more help from home
would want a substitute occasionally and one from the other shift
could pick up a few extra hours of work.
```

Figure A.4: Prose Page (OCR Text) in *Shaver-Dougherty Genealogy*, Page 43.



THE ELY ANCESTRY. 237

SIXTH GENERATION.

13712. Alfred Ely, Ellisburg, N. Y., b. 1788, d. 1870, son of David Ely and Keziah Mapes; m. 1812, Patience Beckwith, Great Barrington, Mass., who was b. 1791, d. 1844, dau. of George Beckwith and Patience Beckwith. Their children:

1. Harriet Beckwith, b. 1813; d. 1815.
2. George Beckwith, b. 1816; d. 1816.
3. George Beckwith, b. 1817; d. 1877; m. 1843, Gertrude Sophia Harmon.
4. Erastus, b. 1819; m. 1844, Adelia Mapes.
5. Emily, b. 1822; d. 1880; m. 1841, Samuel Sedgwick.
6. Harriet Eliza, b. 1830; d. 1830.
7. Alfred, b. 1832 (Euclid Ave., Cleveland, O.); m. 1856, Caroline F. Burnham.
8. Caroline Frances, b. 1837.

13714. Keziah Mapes Ely, West Swansey, N. H., b. 1794, dau. of David Ely and Keziah Mapes; m. 1819, Jotham Eames, West Swansey, N. H., who was b. 1793, d. 1850, son of Jotham Eames and Eusebia Goddard. Their children:

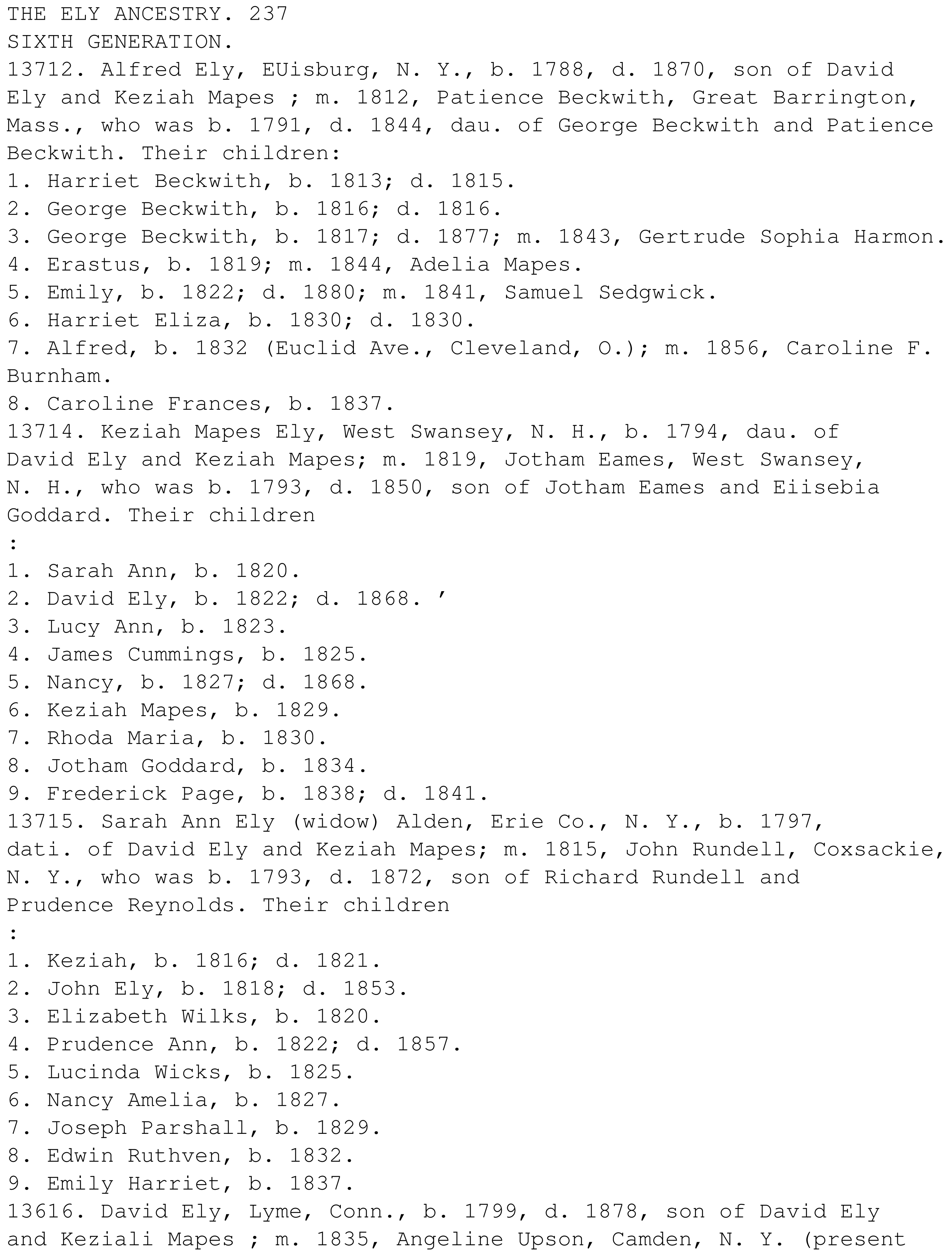
1. Sarah Ann, b. 1820.
2. David Ely, b. 1822; d. 1868.
3. Lucy Ann, b. 1823.
4. James Cummings, b. 1825.
5. Nancy, b. 1827; d. 1868.
6. Keziah Mapes, b. 1829.
7. Rhoda Maria, b. 1830.
8. Jotham Goddard, b. 1834.
9. Frederick Page, b. 1838; d. 1841.

13715. Sarah Ann Ely (widow) Alden, Erie Co., N. Y., b. 1797, dau. of David Ely and Keziah Mapes; m. 1815, John Rundell, Coxsackie, N. Y., who was b. 1793, d. 1872, son of Richard Rundell and Prudence Reynolds. Their children:

1. Keziah, b. 1816; d. 1821.
2. John Ely, b. 1818; d. 1853.
3. Elizabeth Wilks, b. 1820.
4. Prudence Ann, b. 1822; d. 1857.
5. Lucinda Wicks, b. 1825.
6. Nancy Amelia, b. 1827.
7. Joseph Parshall, b. 1829.
8. Edwin Ruthven, b. 1832.
9. Emily Harriet, b. 1837.

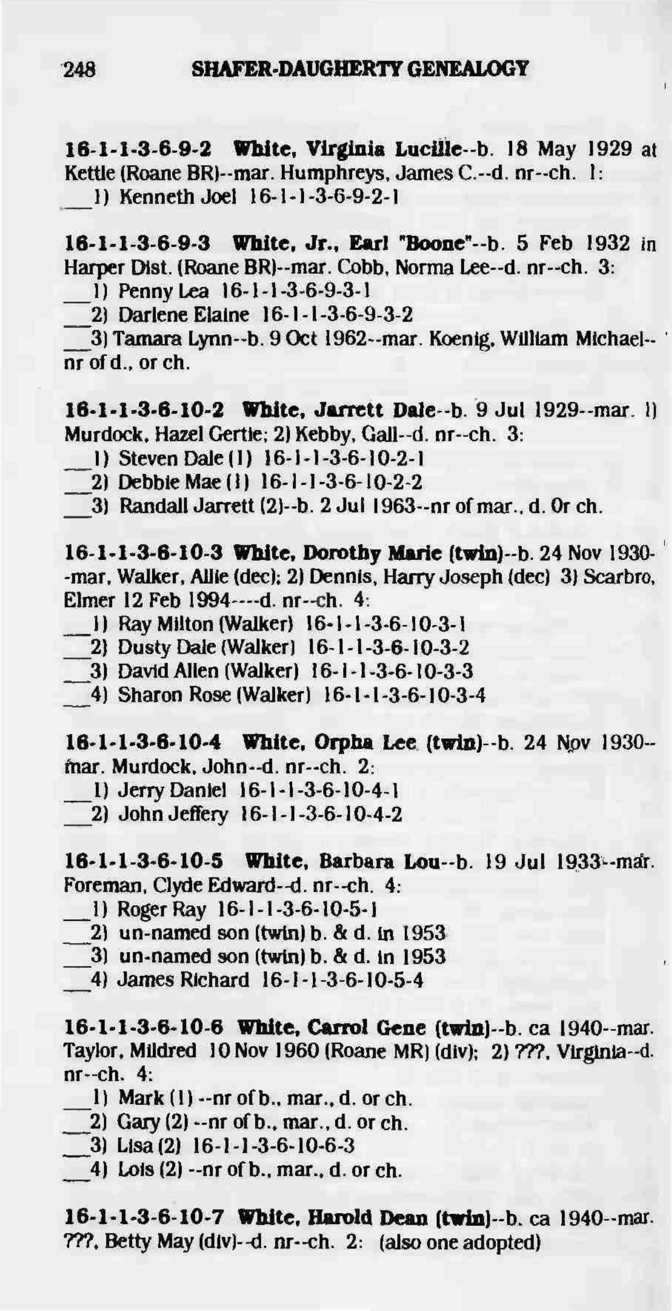
13616. David Ely, Lyme, Conn., b. 1799, d. 1878, son of David Ely and Keziah Mapes; m. 1835, Angeline Upson, Camden, N. Y. (present

Figure A.5: Family List Page in *The Ely Ancestry*, Page 237.



```
THE ELY ANCESTRY. 237
SIXTH GENERATION.
13712. Alfred Ely, EUisburg, N. Y., b. 1788, d. 1870, son of David
Ely and Keziah Mapes ; m. 1812, Patience Beckwith, Great Barrington,
Mass., who was b. 1791, d. 1844, dau. of George Beckwith and Patience
Beckwith. Their children:
1. Harriet Beckwith, b. 1813; d. 1815.
2. George Beckwith, b. 1816; d. 1816.
3. George Beckwith, b. 1817; d. 1877; m. 1843, Gertrude Sophia Harmon.
4. Erastus, b. 1819; m. 1844, Adelia Mapes.
5. Emily, b. 1822; d. 1880; m. 1841, Samuel Sedgwick.
6. Harriet Eliza, b. 1830; d. 1830.
7. Alfred, b. 1832 (Euclid Ave., Cleveland, O.); m. 1856, Caroline F.
Burnham.
8. Caroline Frances, b. 1837.
13714. Keziah Mapes Ely, West Swansey, N. H., b. 1794, dau. of
David Ely and Keziah Mapes; m. 1819, Jotham Eames, West Swansey,
N. H., who was b. 1793, d. 1850, son of Jotham Eames and Eiisebia
Goddard. Their children
:
1. Sarah Ann, b. 1820.
2. David Ely, b. 1822; d. 1868. ’
3. Lucy Ann, b. 1823.
4. James Cummings, b. 1825.
5. Nancy, b. 1827; d. 1868.
6. Keziah Mapes, b. 1829.
7. Rhoda Maria, b. 1830.
8. Jotham Goddard, b. 1834.
9. Frederick Page, b. 1838; d. 1841.
13715. Sarah Ann Ely (widow) Alden, Erie Co., N. Y., b. 1797,
dati. of David Ely and Keziah Mapes; m. 1815, John Rundell, Coxsackie,
N. Y., who was b. 1793, d. 1872, son of Richard Rundell and
Prudence Reynolds. Their children
:
1. Keziah, b. 1816; d. 1821.
2. John Ely, b. 1818; d. 1853.
3. Elizabeth Wilks, b. 1820.
4. Prudence Ann, b. 1822; d. 1857.
5. Lucinda Wicks, b. 1825.
6. Nancy Amelia, b. 1827.
7. Joseph Parshall, b. 1829.
8. Edwin Ruthven, b. 1832.
9. Emily Harriet, b. 1837.
13616. David Ely, Lyme, Conn., b. 1799, d. 1878, son of David Ely
and Keziali Mapes ; m. 1835, Angeline Upson, Camden, N. Y. (present
```

Figure A.6: Family List Page (OCR Text) in *The Ely Ancestry*, Page 237.



248 SHAFER-DAUGHERTY GENEALOGY

16-1-1-3-6-9-2 White, Virginia Lucille--b. 18 May 1929 at Kettle (Roane BR)--mar. Humphreys, James C.--d. nr--ch. 1:
___1) Kenneth Joel 16-1-1-3-6-9-2-1

16-1-1-3-6-9-3 White, Jr., Earl "Boone"--b. 5 Feb 1932 in Harper Dist. (Roane BR)--mar. Cobb, Norma Lee--d. nr--ch. 3:
___1) Penny Lea 16-1-1-3-6-9-3-1
___2) Darlene Elaine 16-1-1-3-6-9-3-2
___3) Tamara Lynn--b. 9 Oct 1962--mar. Koenig, William Michael--nr of d., or ch.

16-1-1-3-6-10-2 White, Jarrett Dale--b. 9 Jul 1929--mar. 1) Murdock, Hazel Gertie; 2) Kebby, Gail--d. nr--ch. 3:
___1) Steven Dale (1) 16-1-1-3-6-10-2-1
___2) Debbie Mae (1) 16-1-1-3-6-10-2-2
___3) Randall Jarrett (2)--b. 2 Jul 1963--nr of mar., d. Or ch.

16-1-1-3-6-10-3 White, Dorothy Marie (twin)--b. 24 Nov 1930--mar, Walker, Allie (dec); 2) Dennis, Harry Joseph (dec) 3) Scarbro, Elmer 12 Feb 1994----d. nr--ch. 4:
___1) Ray Milton (Walker) 16-1-1-3-6-10-3-1
___2) Dusty Dale (Walker) 16-1-1-3-6-10-3-2
___3) David Allen (Walker) 16-1-1-3-6-10-3-3
___4) Sharon Rose (Walker) 16-1-1-3-6-10-3-4

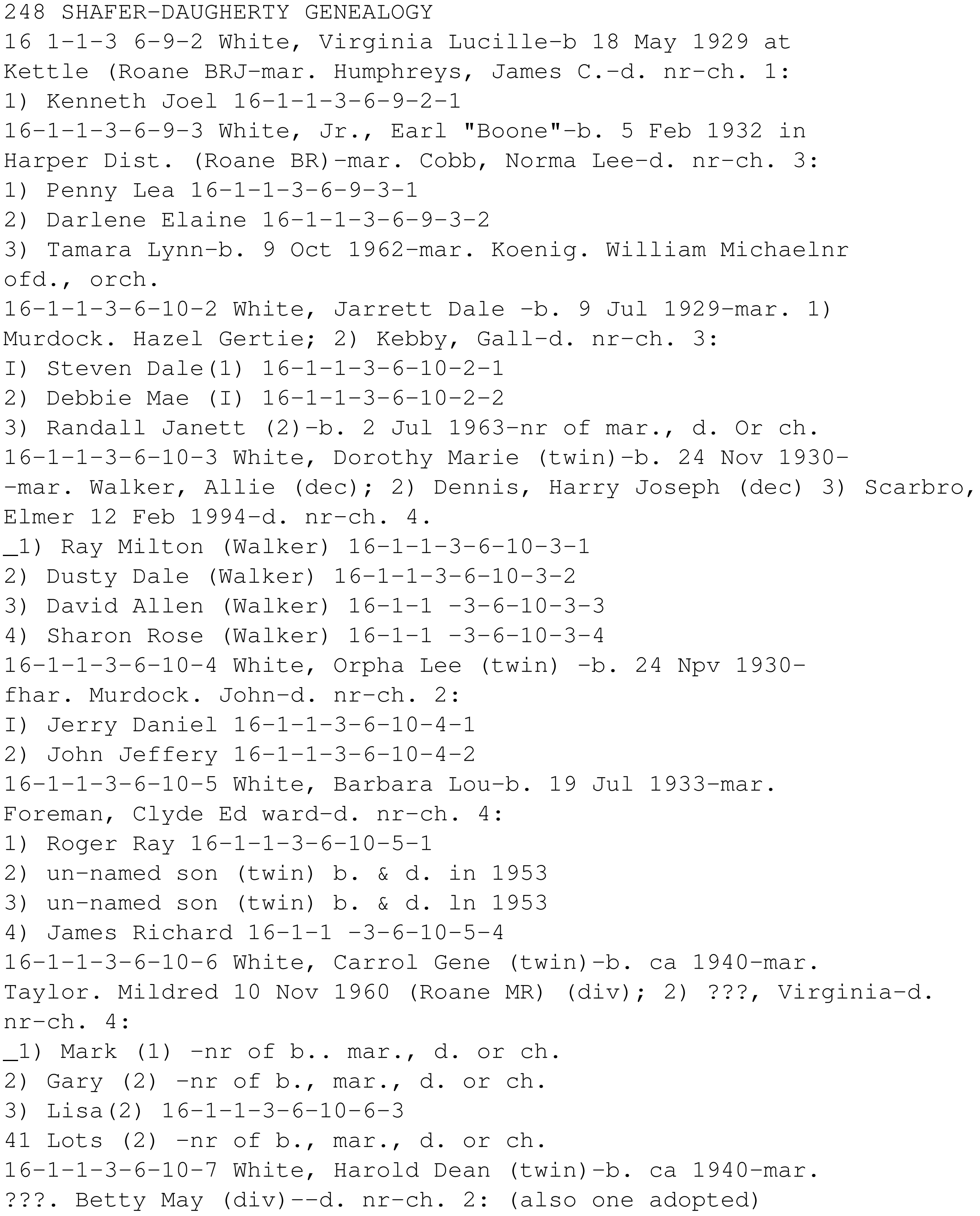
16-1-1-3-6-10-4 White, Orpha Lee (twin)--b. 24 Nov 1930--mar. Murdock, John--d. nr--ch. 2:
___1) Jerry Daniel 16-1-1-3-6-10-4-1
___2) John Jeffery 16-1-1-3-6-10-4-2

16-1-1-3-6-10-5 White, Barbara Lou--b. 19 Jul 1933--mar. Foreman, Clyde Edward--d. nr--ch. 4:
___1) Roger Ray 16-1-1-3-6-10-5-1
___2) un-named son (twin) b. & d. in 1953
___3) un-named son (twin) b. & d. in 1953
___4) James Richard 16-1-1-3-6-10-5-4

16-1-1-3-6-10-6 White, Carrol Gene (twin)--b. ca 1940--mar. Taylor, Mildred 10 Nov 1960 (Roane MR) (div); 2) ???, Virginia--d. nr--ch. 4:
___1) Mark (1) --nr of b., mar., d. or ch.
___2) Gary (2) --nr of b., mar., d. or ch.
___3) Lisa (2) 16-1-1-3-6-10-6-3
___4) Lois (2) --nr of b., mar., d. or ch.

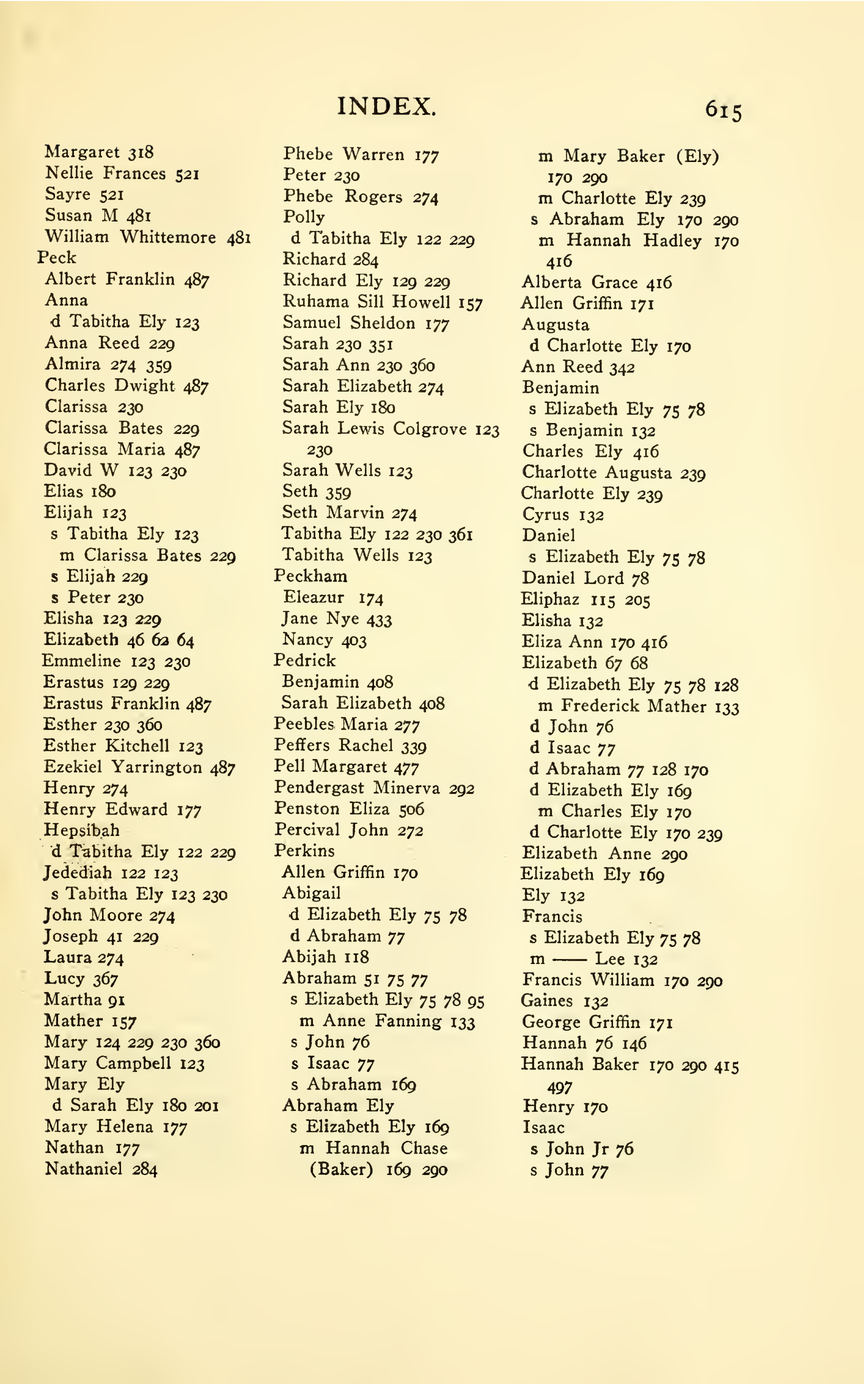
16-1-1-3-6-10-7 White, Harold Dean (twin)--b. ca 1940--mar. ???, Betty May (div)--d. nr--ch. 2: (also one adopted)

Figure A.7: Family List Page in *Shaver-Dougherty Genealogy*, Page 248.



```
248 SHAFER-DAUGHERTY GENEALOGY
16 1-1-3 6-9-2 White, Virginia Lucille-b 18 May 1929 at
Kettle (Roane BRJ-mar. Humphreys, James C.-d. nr-ch. 1:
1) Kenneth Joel 16-1-1-3-6-9-2-1
16-1-1-3-6-9-3 White, Jr., Earl "Boone"-b. 5 Feb 1932 in
Harper Dist. (Roane BR)-mar. Cobb, Norma Lee-d. nr-ch. 3:
1) Penny Lea 16-1-1-3-6-9-3-1
2) Darlene Elaine 16-1-1-3-6-9-3-2
3) Tamara Lynn-b. 9 Oct 1962-mar. Koenig. William Michaelnr
ofd., orch.
16-1-1-3-6-10-2 White, Jarrett Dale -b. 9 Jul 1929-mar. 1)
Murdock. Hazel Gertie; 2) Kebby, Gall-d. nr-ch. 3:
I) Steven Dale(1) 16-1-1-3-6-10-2-1
2) Debbie Mae (I) 16-1-1-3-6-10-2-2
3) Randall Janett (2)-b. 2 Jul 1963-nr of mar., d. Or ch.
16-1-1-3-6-10-3 White, Dorothy Marie (twin)-b. 24 Nov 1930-
-mar. Walker, Allie (dec); 2) Dennis, Harry Joseph (dec) 3) Scarbro,
Elmer 12 Feb 1994-d. nr-ch. 4.
_1) Ray Milton (Walker) 16-1-1-3-6-10-3-1
2) Dusty Dale (Walker) 16-1-1-3-6-10-3-2
3) David Allen (Walker) 16-1-1 -3-6-10-3-3
4) Sharon Rose (Walker) 16-1-1 -3-6-10-3-4
16-1-1-3-6-10-4 White, Orpha Lee (twin) -b. 24 Npv 1930-
fhar. Murdock. John-d. nr-ch. 2:
I) Jerry Daniel 16-1-1-3-6-10-4-1
2) John Jeffery 16-1-1-3-6-10-4-2
16-1-1-3-6-10-5 White, Barbara Lou-b. 19 Jul 1933-mar.
Foreman, Clyde Ed ward-d. nr-ch. 4:
1) Roger Ray 16-1-1-3-6-10-5-1
2) un-named son (twin) b. & d. in 1953
3) un-named son (twin) b. & d. ln 1953
4) James Richard 16-1-1 -3-6-10-5-4
16-1-1-3-6-10-6 White, Carrol Gene (twin)-b. ca 1940-mar.
Taylor. Mildred 10 Nov 1960 (Roane MR) (div); 2) ???, Virginia-d.
nr-ch. 4:
_1) Mark (1) -nr of b.. mar., d. or ch.
2) Gary (2) -nr of b., mar., d. or ch.
3) Lisa(2) 16-1-1-3-6-10-6-3
41 Lots (2) -nr of b., mar., d. or ch.
16-1-1-3-6-10-7 White, Harold Dean (twin)-b. ca 1940-mar.
???. Betty May (div)--d. nr-ch. 2: (also one adopted)
```

Figure A.8: Family List Page (OCR Text) in *Shaver-Dougherty Genealogy*, Page 248.



INDEX. 615

Margaret 318
Nellie Frances 521
Sayre 521
Susan M 481
William Whittemore 481
Peck
Albert Franklin 487
Anna
d Tabitha Ely 123
Anna Reed 229
Almira 274 359
Charles Dwight 487
Clarissa 230
Clarissa Bates 229
Clarissa Maria 487
David W 123 230
Elias 180
Elijah 123
s Tabitha Ely 123
m Clarissa Bates 229
s Elijah 229
s Peter 230
Elisha 123 229
Elizabeth 46 62 64
Emmeline 123 230
Erastus 129 229
Erastus Franklin 487
Esther 230 360
Esther Kitchell 123
Ezekiel Yarrington 487
Henry 274
Henry Edward 177
Hepsibah
d Tabitha Ely 122 229
Jedediah 122 123
s Tabitha Ely 123 230
John Moore 274
Joseph 41 229
Laura 274
Lucy 367
Martha 91
Mather 157
Mary 124 229 230 360
Mary Campbell 123
Mary Ely
d Sarah Ely 180 201
Mary Helena 177
Nathan 177
Nathaniel 284

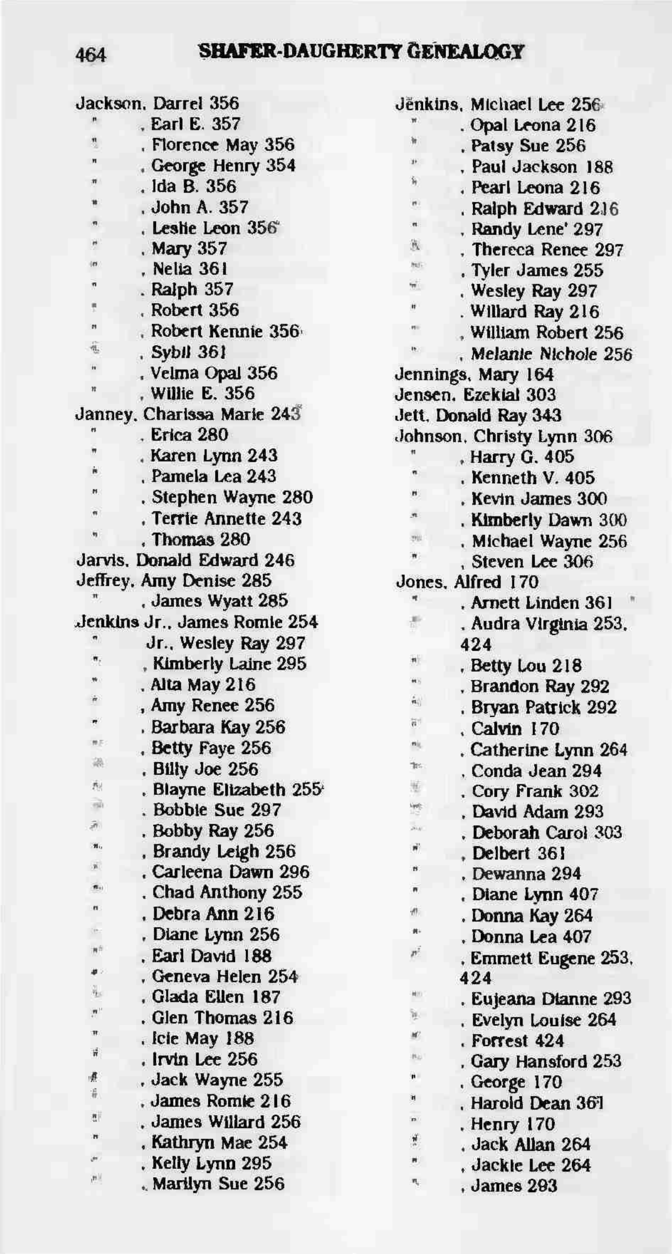
Phebe Warren 177
Peter 230
Phebe Rogers 274
Polly
d Tabitha Ely 122 229
Richard 284
Richard Ely 129 229
Ruhama Sill Howell 157
Samuel Sheldon 177
Sarah 230 351
Sarah Ann 230 360
Sarah Elizabeth 274
Sarah Ely 180
Sarah Lewis Colgrove 123 230
Sarah Wells 123
Seth 359
Seth Marvin 274
Tabitha Ely 122 230 361
Tabitha Wells 123
Peckham
Eleazur 174
Jane Nye 433
Nancy 403
Pedrick
Benjamin 408
Sarah Elizabeth 408
Peebles Maria 277
Peffers Rachel 339
Pell Margaret 477
Pendergast Minerva 292
Penston Eliza 506
Percival John 272
Perkins
Allen Griffin 170
Abigail
d Elizabeth Ely 75 78
d Abraham 77
Abijah 118
Abraham 51 75 77
s Elizabeth Ely 75 78 95
m Anne Fanning 133
s John 76
s Isaac 77
s Abraham 169
Abraham Ely
s Elizabeth Ely 169
m Hannah Chase (Baker) 169 290

m Mary Baker (Ely) 170 290
m Charlotte Ely 239
s Abraham Ely 170 290
m Hannah Hadley 170 416
Alberta Grace 416
Allen Griffin 171
Augusta
d Charlotte Ely 170
Ann Reed 342
Benjamin
s Elizabeth Ely 75 78
s Benjamin 132
Charles Ely 416
Charlotte Augusta 239
Charlotte Ely 239
Cyrus 132
Daniel
s Elizabeth Ely 75 78
Daniel Lord 78
Eliphaz 115 205
Elisha 132
Eliza Ann 170 416
Elizabeth 67 68
d Elizabeth Ely 75 78 128
m Frederick Mather 133
d John 76
d Isaac 77
d Abraham 77 128 170
d Elizabeth Ely 169
m Charles Ely 170
d Charlotte Ely 170 239
Elizabeth Anne 290
Elizabeth Ely 169
Ely 132
Francis
s Elizabeth Ely 75 78
m —— Lee 132
Francis William 170 290
Gaines 132
George Griffin 171
Hannah 76 146
Hannah Baker 170 290 415 497
Henry 170
Isaac
s John Jr 76
s John 77

Figure A.9: Name Index Page in *The Ely Ancestry*, Page 615.

```
INDEX. 615
Margaret 318
Nellie Frances 521
Sayre 521
Susan M 481
William Whittemore 481
Peck
Albert Franklin 487
...
Perkins
Allen Grififin 170
Abigail
d Elizabeth Ely 75 78
d Abraham 'J^
Abijah 118
Abraham 51 75 yy
s Elizabeth Ely 75 78 95
m Anne Fanning 133
s John ^(>
s Isaac ^^
s Abraham 169
Abraham, Ely
s Elizabeth Ely 169
m Hannah Chase
(Baker) 169 290
...
Elizabeth (i-j 68
d Elizabeth Ely 75 78 128
m Frederick Mather 133
d John 76
d Isaac Ty
d Abraham ']^ 128 170
d Elizabeth Ely 169
m Charles Ely 170
d Charlotte Ely 170 239
...
Hannah 76 146
Hannah Baker 170 290 415
497
Henry 170
Isaac
s John Jr "jd
s John TJ
```

Figure A.10: Name Index Page (OCR Text) in *The Ely Ancestry*, Page 615.



464 SHAFER-DAUGHERTY GENEALOGY

Jackson, Darrel 356
" , Earl E. 357
" , Florence May 356
" , George Henry 354
" , Ida B. 356
" , John A. 357
" , Leslie Leon 356
" , Mary 357
" , Nelia 361
" , Ralph 357
" , Robert 356
" , Robert Kennie 356
" , Sybil 361
" , Velma Opal 356
" , Willie E. 356
Janney, Charissa Marie 243
" , Erica 280
" , Karen Lynn 243
" , Pamela Lea 243
" , Stephen Wayne 280
" , Terrie Annette 243
" , Thomas 280
Jarvis, Donald Edward 246
Jeffrey, Amy Denise 285
" , James Wyatt 285
Jenkins Jr., James Romie 254
" Jr., Wesley Ray 297
" , Kimberly Laine 295
" , Alta May 216
" , Amy Renee 256
" , Barbara Kay 256
" , Betty Faye 256
" , Billy Joe 256
" , Blayne Elizabeth 255
" , Bobbie Sue 297
" , Bobby Ray 256
" , Brandy Leigh 256
" , Carleena Dawn 296
" , Chad Anthony 255
" , Debra Ann 216
" , Diane Lynn 256
" , Earl David 188
" , Geneva Helen 254
" , Glada Ellen 187
" , Glen Thomas 216
" , Icie May 188
" , Irvin Lee 256
" , Jack Wayne 255
" , James Romie 216
" , James Willard 256
" , Kathryn Mae 254
" , Kelly Lynn 295
" , Marilyn Sue 256

Jenkins, Michael Lee 256
" , Opal Leona 216
" , Patsy Sue 256
" , Paul Jackson 188
" , Pearl Leona 216
" , Ralph Edward 216
" , Randy Lene' 297
" , Thereca Renee 297
" , Tyler James 255
" , Wesley Ray 297
" , Willard Ray 216
" , William Robert 256
" , Melanie Nichole 256
Jennings, Mary 164
Jensen, Ezekial 303
Jett, Donald Ray 343
Johnson, Christy Lynn 306
" , Harry G. 405
" , Kenneth V. 405
" , Kevin James 300
" , Kimberly Dawn 300
" , Michael Wayne 256
" , Steven Lee 306
Jones, Alfred 170
" , Arnett Linden 361
" , Audra Virginia 253, 424
" , Betty Lou 218
" , Brandon Ray 292
" , Bryan Patrick 292
" , Calvin 170
" , Catherine Lynn 264
" , Conda Jean 294
" , Cory Frank 302
" , David Adam 293
" , Deborah Carol 303
" , Delbert 361
" , Dewanna 294
" , Diane Lynn 407
" , Donna Kay 264
" , Donna Lea 407
" , Emmett Eugene 253, 424
" , Eujeana Dianne 293
" , Evelyn Louise 264
" , Forrest 424
" , Gary Hansford 253
" , George 170
" , Harold Dean 361
" , Henry 170
" , Jack Allan 264
" , Jackie Lee 264
" , James 293

Figure A.11: Name Index Page in *Shaver-Dougherty Genealogy*, Page 464.

```
464 SHAFER-DAUGHERTY GENEALOGY
Jackson, Darrel 356
, Earl E. 357
. Florence May 356
, George Henry 354
. Ida B. 356
. John A. 357
. Leslie Leon 356*
. Mary 357
,Nelia361
. Ralph 357
. Robert 356
. Robert Kennie 35fr
. Sybil 361
...
, Diane Lynn 256
. Earl David 188
* , Geneva Helen 254
. Glada EUen 187
. Glen Thomas 216
. Icie May 188
. Irvin Lee 256
'* , Jack Wayne 255
.James Romie 216
. James Willard 256
. Kathryn Mae 254
. Kelly Lynn 295
.. Marilyn Sue 256
Jenkins. Michael Lee 256
.Opal Leona 216
. Patsy Sue 256
, Paul Jackson 188
. Pearl Leona 216
. Ralph Edward 2,16
, Randy Lene' 297
* , Thereca Renee 297
...
. George 170
. Harold Dean 36 _
.Henry 170
. Jack Allan 264
, Jackie Lee 264
, James 293
```

Figure A.12: Name Index Page (OCR Text) in *Shaver-Dougherty Genealogy*, Page 464.

Appendix B

Justification of *pattern-length*×*frequency-of-occurrence* for Scoring Clusters

We acknowledge a trade-off between looking for highly frequent patterns to improve labeling efficiency and recall on the one hand, and looking for longer patterns to improve accuracy or precision on the other. We balance these requirements using an asymptotic simplification of the usual minimum description length (MDL) formula used in types of grammar induction based on information theory and information compression. This simplified formula (the product of length and frequency) allows ListReader to select record templates to insert as phrase structure rules into its grammar with good precision and recall. In Equation B.1, we derive the simplified formula ListReader uses to measure the benefit of adding a production rule $G_{lhs} \rightarrow G_{rhs}$ to grammar G and replacing all occurrences of its right hand side n-gram in text X with its left hand side symbol. (G_{lhs} represents a whole record and G_{rhs} represents its constituents which are a sequence of conflated text.)

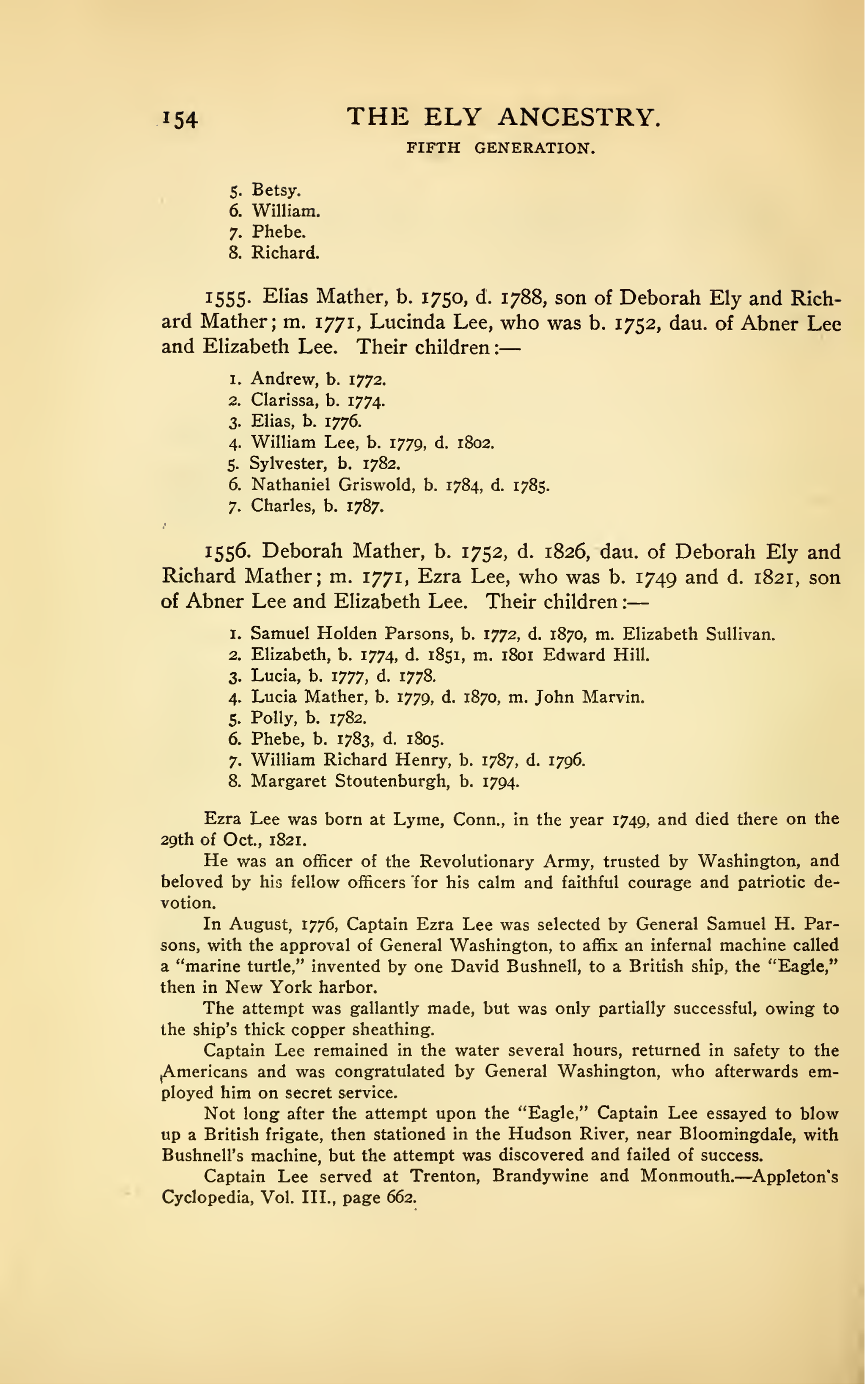
$$\begin{aligned}
DL(X, G) &= DL(G) + DL(X|G) \\
&= \sum_{g_{rhs} \in G_{rhs}} DL(g_{rhs}) + DL(X) - \sum_{g_{rhs} \in G_{rhs}} c(g_{rhs})DL(g_{rhs}) + c(g_{lhs})DL(g_{lhs}) \\
&\propto \sum_{g_{rhs} \in G_{rhs}} DL(g_{rhs}) - \sum_{g_{rhs} \in G_{rhs}} c(g_{rhs})DL(g_{rhs}) + c(g_{lhs})DL(g_{lhs}) \\
&\approx len(G_{rhs}) - len(G_{rhs})c(G_{rhs}) + c(G_{lhs}) \\
&\approx -len(G_{rhs})c(G_{rhs})
\end{aligned} \tag{B.1}$$

Note that $len(G_{rhs})$ is the length of the right hand side of the proposed grammar rule G and $c(g)$ is the frequency or count of a symbol g within input text X. Note also that the description length of X, $DL(X)$, is the same for all proposed rules, so it does not affect the selection of rules. We have two simplifying assumptions. First, the description length of a symbol g, $DL(g) = -\log p(g)$, is fixed for all symbols and equal to 1. Second, we remove all but the highest-order term as we do in complexity analysis. Selecting record patterns with this simplified formula is fast, is supported directly by the suffix tree, and induces a grammar that effectively "compresses" the text by identifying the natural structure of lists. It in turn reduces annotation cost by allowing ListReader to be selective in requesting labels from the user. Its sensitivity to pattern frequency enables ListReader to maximize the value of even the first requested label from the user and to eliminate queries that have little applicability. Its sensitivity to pattern length improves ListReader's accuracy. Because of work on information compression and machine learning since Solomonov [55], we understand that it is not coincidental that this compressibility should improve learning because of the mathematical correspondence between MDL and maximum a posteriori (MAP) model selection. Roughly speaking, high values of a pattern's length times its frequency is correlated with high likelihood, while low values of a pattern's length alone is correlated with high prior probability. Since length alone and frequency alone are lower-order terms, we ignore them while noting that low values of either would be compensated for by high values in the likelihood term as soon as the length or the frequency is above 2. The record-selection parameter input to ListReader described in Subsection 4.5.1 guarantees that this threshold is met.

Appendix C

Example Pages from Ely, Shaver, and Kilbarchan

We now show an example page from *The Ely Ancestry* in Figure C.1, from the *Shaver-Dougherty Genealogy* in Figure C.2, and three pages from the *Kilbarchan Parish Register* in Figures C.3, C.4, and C.5.



FIFTH GENERATION.

5. Betsy.
6. William.
7. Phebe.
8. Richard.

1555. Elias Mather, b. 1750, d. 1788, son of Deborah Ely and Richard Mather; m. 1771, Lucinda Lee, who was b. 1752, dau. of Abner Lee and Elizabeth Lee. Their children:—

1. Andrew, b. 1772.
2. Clarissa, b. 1774.
3. Elias, b. 1776.
4. William Lee, b. 1779, d. 1802.
5. Sylvester, b. 1782.
6. Nathaniel Griswold, b. 1784, d. 1785.
7. Charles, b. 1787.

1556. Deborah Mather, b. 1752, d. 1826, dau. of Deborah Ely and Richard Mather; m. 1771, Ezra Lee, who was b. 1749 and d. 1821, son of Abner Lee and Elizabeth Lee. Their children:—

1. Samuel Holden Parsons, b. 1772, d. 1870, m. Elizabeth Sullivan.
2. Elizabeth, b. 1774, d. 1851, m. 1801 Edward Hill.
3. Lucia, b. 1777, d. 1778.
4. Lucia Mather, b. 1779, d. 1870, m. John Marvin.
5. Polly, b. 1782.
6. Phebe, b. 1783, d. 1805.
7. William Richard Henry, b. 1787, d. 1796.
8. Margaret Stoutenburgh, b. 1794.

Ezra Lee was born at Lyme, Conn., in the year 1749, and died there on the 29th of Oct., 1821.

He was an officer of the Revolutionary Army, trusted by Washington, and beloved by his fellow officers for his calm and faithful courage and patriotic devotion.

In August, 1776, Captain Ezra Lee was selected by General Samuel H. Parsons, with the approval of General Washington, to affix an infernal machine called a "marine turtle," invented by one David Bushnell, to a British ship, the "Eagle," then in New York harbor.

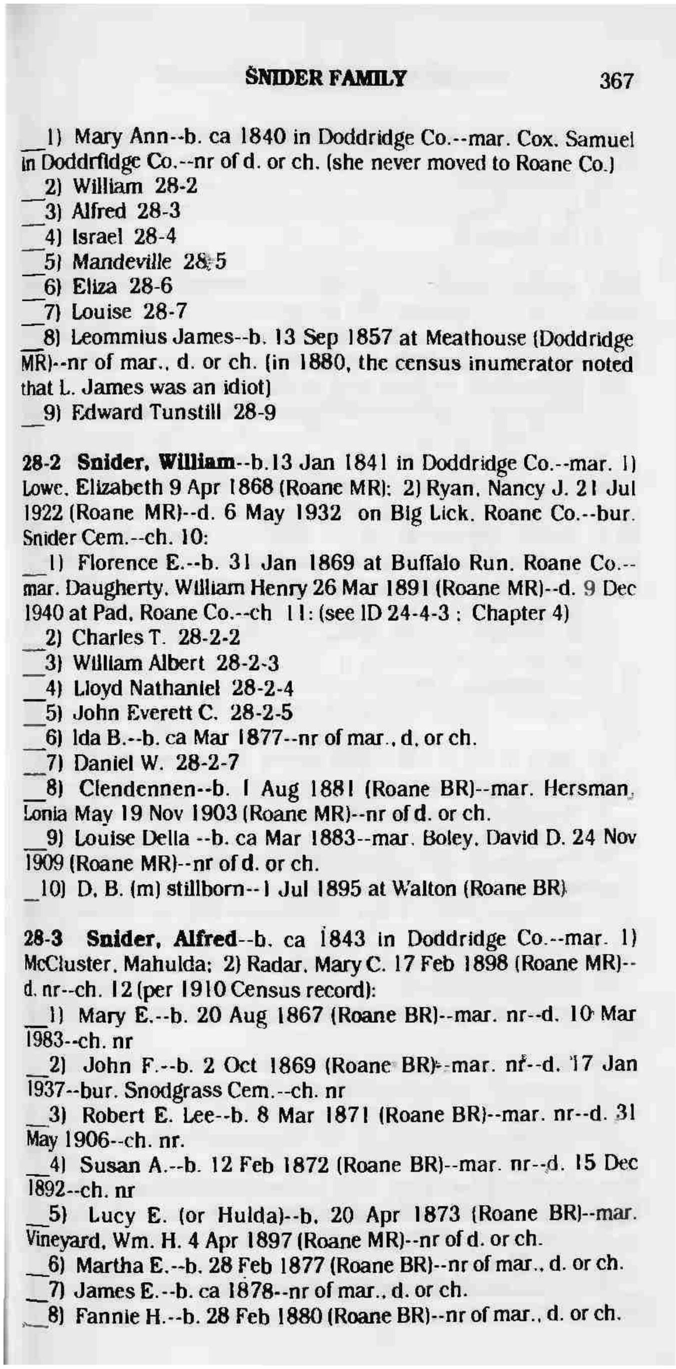
The attempt was gallantly made, but was only partially successful, owing to the ship's thick copper sheathing.

Captain Lee remained in the water several hours, returned in safety to the Americans and was congratulated by General Washington, who afterwards employed him on secret service.

Not long after the attempt upon the "Eagle," Captain Lee essayed to blow up a British frigate, then stationed in the Hudson River, near Bloomingdale, with Bushnell's machine, but the attempt was discovered and failed of success.

Captain Lee served at Trenton, Brandywine and Monmouth.—Appleton's Cyclopedia, Vol. III., page 662.

Figure C.1: Page from *The Ely Ancestry*, Page 367.



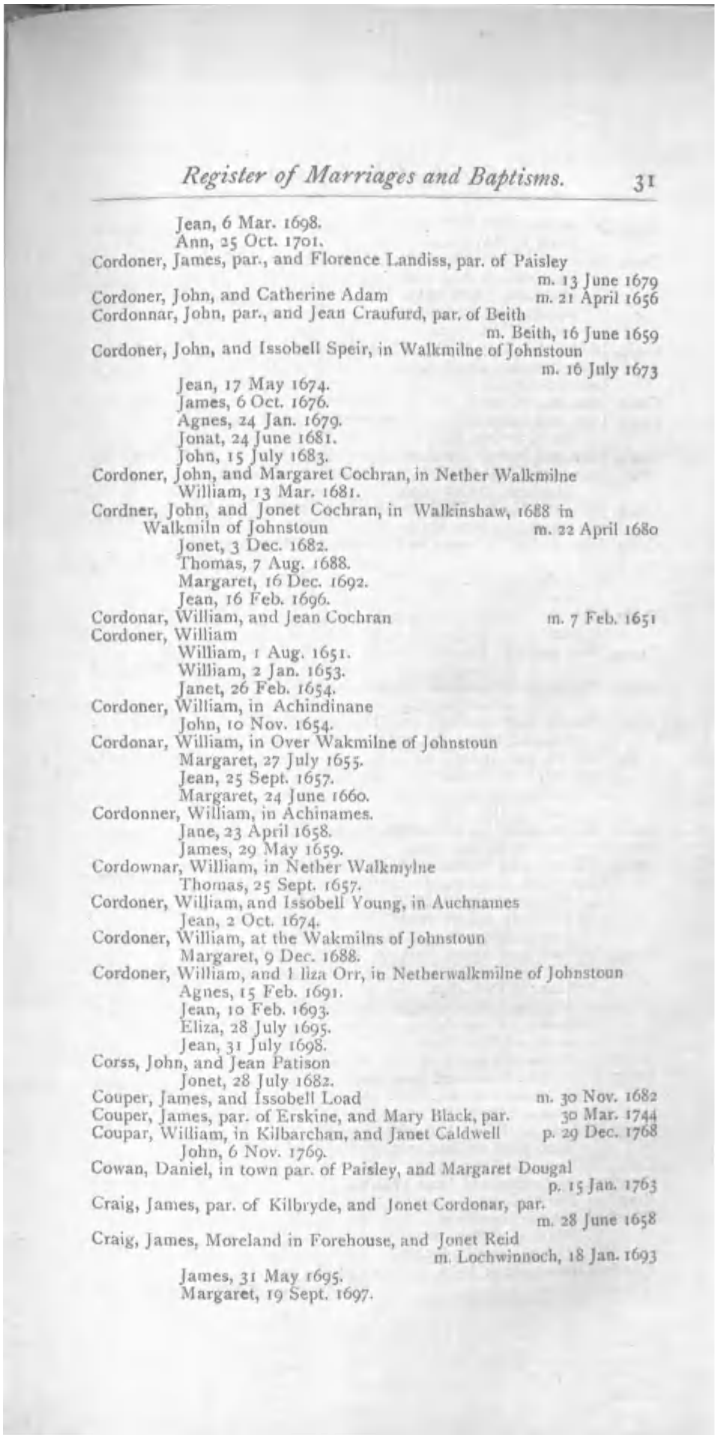
ŠNIDER FAMILY 367

__1) Mary Ann--b. ca 1840 in Doddridge Co.--mar. Cox. Samuel in Doddrfidge Co.--nr of d. or ch. (she never moved to Roane Co.)
__2) William 28-2
__3) Alfred 28-3
__4) Israel 28-4
__5) Mandeville 28-5
__6) Eliza 28-6
__7) Louise 28-7
__8) Leommius James--b. 13 Sep 1857 at Meathouse (Doddridge MR)--nr of mar., d. or ch. (in 1880, the census inumerator noted that L. James was an idiot)
__9) Edward Tunstill 28-9

28-2 **Snider, William**--b.13 Jan 1841 in Doddridge Co.--mar. 1) Lowe, Elizabeth 9 Apr 1868 (Roane MR); 2) Ryan, Nancy J. 21 Jul 1922 (Roane MR)--d. 6 May 1932 on Big Lick, Roane Co.--bur. Snider Cem.--ch. 10:
__1) Florence E.--b. 31 Jan 1869 at Buffalo Run, Roane Co.--mar. Daugherty, William Henry 26 Mar 1891 (Roane MR)--d. 9 Dec 1940 at Pad, Roane Co.--ch 11: (see ID 24-4-3 ; Chapter 4)
__2) Charles T. 28-2-2
__3) William Albert 28-2-3
__4) Lloyd Nathaniel 28-2-4
__5) John Everett C. 28-2-5
__6) Ida B.--b. ca Mar 1877--nr of mar., d. or ch.
__7) Daniel W. 28-2-7
__8) Clendennen--b. 1 Aug 1881 (Roane BR)--mar. Hersman, Lonia May 19 Nov 1903 (Roane MR)--nr of d. or ch.
__9) Louise Della --b. ca Mar 1883--mar. Boley, David D. 24 Nov 1909 (Roane MR)--nr of d. or ch.
__10) D. B. (m) stillborn--1 Jul 1895 at Walton (Roane BR)

28-3 **Snider, Alfred**--b. ca 1843 in Doddridge Co.--mar. 1) McCluster, Mahulda; 2) Radar, Mary C. 17 Feb 1898 (Roane MR)--d. nr--ch. 12 (per 1910 Census record):
__1) Mary E.--b. 20 Aug 1867 (Roane BR)--mar. nr--d. 10 Mar 1983--ch. nr
__2) John F.--b. 2 Oct 1869 (Roane BR)--mar. nr--d. 17 Jan 1937--bur. Snodgrass Cem.--ch. nr
__3) Robert E. Lee--b. 8 Mar 1871 (Roane BR)--mar. nr--d. 31 May 1906--ch. nr.
__4) Susan A.--b. 12 Feb 1872 (Roane BR)--mar. nr--d. 15 Dec 1892--ch. nr
__5) Lucy E. (or Hulda)--b. 20 Apr 1873 (Roane BR)--mar. Vineyard, Wm. H. 4 Apr 1897 (Roane MR)--nr of d. or ch.
__6) Martha E.--b. 28 Feb 1877 (Roane BR)--nr of mar., d. or ch.
__7) James E.--b. ca 1878--nr of mar., d. or ch.
__8) Fannie H.--b. 28 Feb 1880 (Roane BR)--nr of mar., d. or ch.

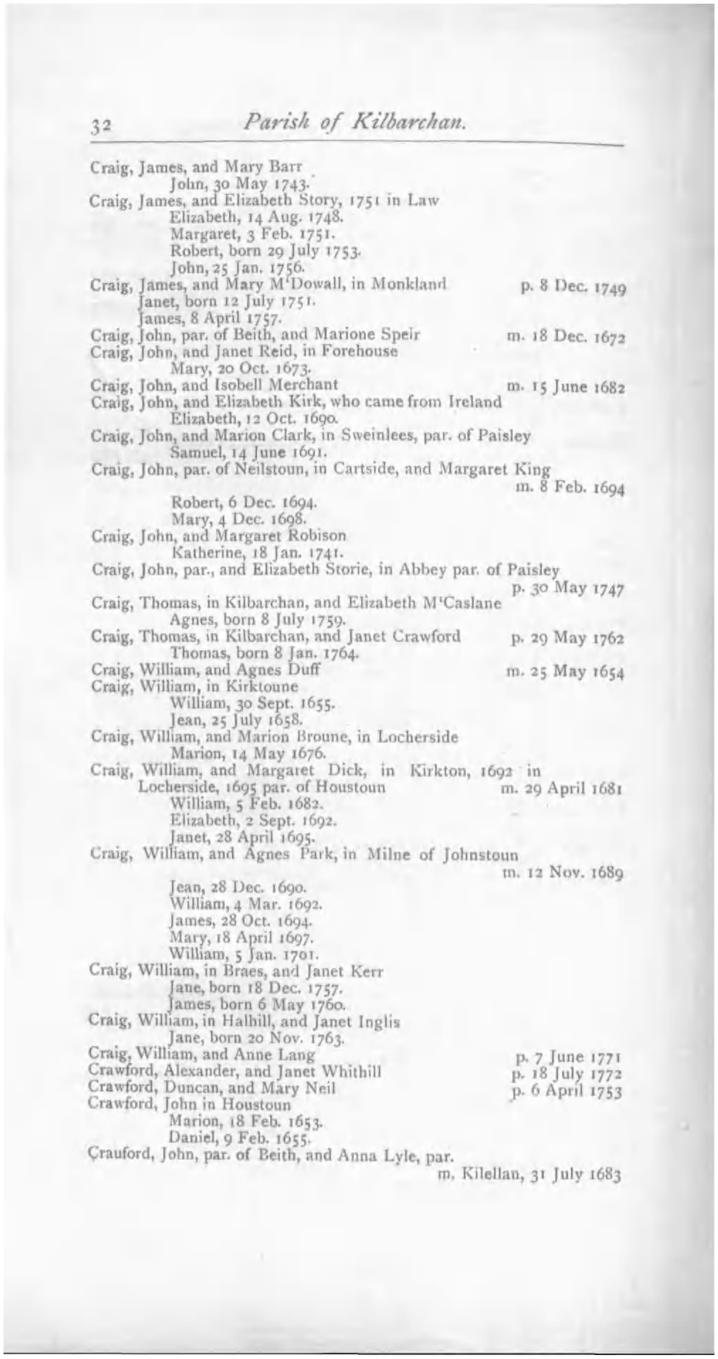
Figure C.2: Page from *Shaver-Dougherty Genealogy*, Page 154.



Register of Marriages and Baptisms. 31

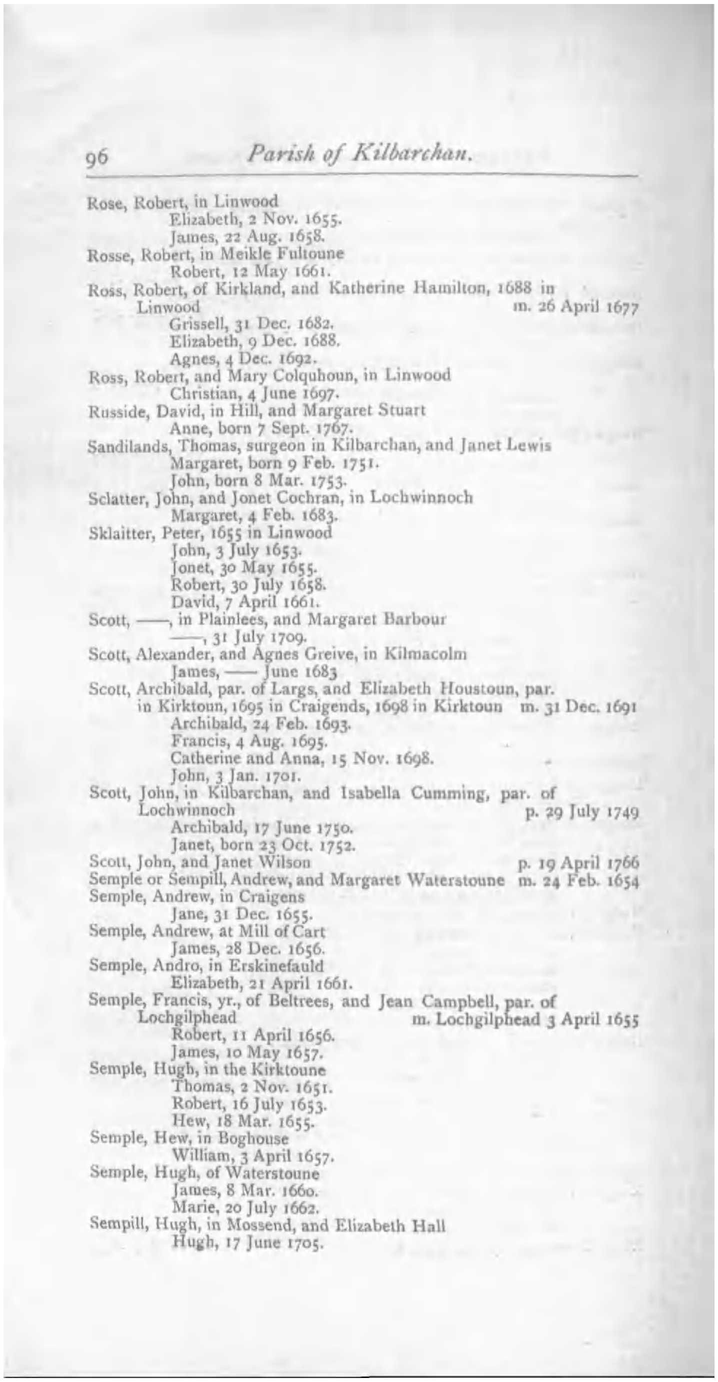
Jean, 6 Mar. 1698.
Ann, 25 Oct. 1701.
Cordoner, James, par., and Florence Landiss, par. of Paisley
m. 13 June 1679
Cordoner, John, and Catherine Adam m. 21 April 1656
Cordonnar, John, par., and Jean Craufurd, par. of Beith
m. Beith, 16 June 1659
Cordoner, John, and Issobell Speir, in Walkmilne of Johnstoun
m. 16 July 1673
Jean, 17 May 1674.
James, 6 Oct. 1676.
Agnes, 24 Jan. 1679.
Jonat, 24 June 1681.
John, 15 July 1683.
Cordoner, John, and Margaret Cochran, in Nether Walkmilne
William, 13 Mar. 1681.
Cordner, John, and Jonet Cochran, in Walkinshaw, 1688 in
Walkmiln of Johnstoun m. 22 April 1680
Jonet, 3 Dec. 1682.
Thomas, 7 Aug. 1688.
Margaret, 16 Dec. 1692.
Jean, 16 Feb. 1696.
Cordonar, William, and Jean Cochran m. 7 Feb. 1651
Cordoner, William
William, 1 Aug. 1651.
William, 2 Jan. 1653.
Janet, 26 Feb. 1654.
Cordoner, William, in Achindinane
John, 10 Nov. 1654.
Cordonar, William, in Over Wakmilne of Johnstoun
Margaret, 27 July 1655.
Jean, 25 Sept. 1657.
Margaret, 24 June 1660.
Cordonner, William, in Achinames.
Jane, 23 April 1658.
James, 29 May 1659.
Cordownar, William, in Nether Walkmylne
Thomas, 25 Sept. 1657.
Cordoner, William, and Issobell Young, in Auchnames
Jean, 2 Oct. 1674.
Cordoner, William, at the Wakmilns of Johnstoun
Margaret, 9 Dec. 1688.
Cordoner, William, and I liza Orr, in Netherwalkmilne of Johnstoun
Agnes, 15 Feb. 1691.
Jean, 10 Feb. 1693.
Eliza, 28 July 1695.
Jean, 31 July 1698.
Corss, John, and Jean Patison
Jonet, 28 July 1682.
Couper, James, and Issobell Load m. 30 Nov. 1682
Couper, James, par. of Erskine, and Mary Black, par. 30 Mar. 1744
Coupar, William, in Kilbarchan, and Janet Caldwell p. 29 Dec. 1768
John, 6 Nov. 1769.
Cowan, Daniel, in town par. of Paisley, and Margaret Dougal
p. 15 Jan. 1763
Craig, James, par. of Kilbryde, and Jonet Cordonar, par.
m. 28 June 1658
Craig, James, Moreland in Forehouse, and Jonet Reid
m. Lochwinnoch, 18 Jan. 1693
James, 31 May 1695.
Margaret, 19 Sept. 1697.

Figure C.3: Page from *Kilbarchan Parish Register*, Page 31.



Craig, James, and Mary Barr
John, 30 May 1743.
Craig, James, and Elizabeth Story, 1751 in Law
Elizabeth, 14 Aug. 1748.
Margaret, 3 Feb. 1751.
Robert, born 29 July 1753.
John, 25 Jan. 1756.
Craig, James, and Mary M'Dowall, in Monkland p. 8 Dec. 1749
Janet, born 12 July 1751.
James, 8 April 1757.
Craig, John, par. of Beith, and Marione Speir m. 18 Dec. 1672
Craig, John, and Janet Reid, in Forehouse
Mary, 20 Oct. 1673.
Craig, John, and Isobell Merchant m. 15 June 1682
Craig, John, and Elizabeth Kirk, who came from Ireland
Elizabeth, 12 Oct. 1690.
Craig, John, and Marion Clark, in Sweinlees, par. of Paisley
Samuel, 14 June 1691.
Craig, John, par. of Neilstoun, in Cartside, and Margaret King
m. 8 Feb. 1694
Robert, 6 Dec. 1694.
Mary, 4 Dec. 1698.
Craig, John, and Margaret Robison
Katherine, 18 Jan. 1741.
Craig, John, par., and Elizabeth Storie, in Abbey par. of Paisley
p. 30 May 1747
Craig, Thomas, in Kilbarchan, and Elizabeth M'Caslane
Agnes, born 8 July 1759.
Craig, Thomas, in Kilbarchan, and Janet Crawford p. 29 May 1762
Thomas, born 8 Jan. 1764.
Craig, William, and Agnes Duff m. 25 May 1654
Craig, William, in Kirktoune
William, 30 Sept. 1655.
Jean, 25 July 1658.
Craig, William, and Marion Broune, in Locherside
Marion, 14 May 1676.
Craig, William, and Margaret Dick, in Kirkton, 1692 in
Locherside, 1695 par. of Houstoun m. 29 April 1681
William, 5 Feb. 1682.
Elizabeth, 2 Sept. 1692.
Janet, 28 April 1695.
Craig, William, and Agnes Park, in Milne of Johnstoun
m. 12 Nov. 1689
Jean, 28 Dec. 1690.
William, 4 Mar. 1692.
James, 28 Oct. 1694.
Mary, 18 April 1697.
William, 5 Jan. 1701.
Craig, William, in Braes, and Janet Kerr
Jane, born 18 Dec. 1757.
James, born 6 May 1760.
Craig, William, in Halhill, and Janet Inglis
Jane, born 20 Nov. 1763.
Craig, William, and Anne Lang p. 7 June 1771
Crawford, Alexander, and Janet Whithill p. 18 July 1772
Crawford, Duncan, and Mary Neil p. 6 April 1753
Crawford, John in Houstoun
Marion, 18 Feb. 1653.
Daniel, 9 Feb. 1655.
Çrauford, John, par. of Beith, and Anna Lyle, par.
m. Kilellan, 31 July 1683

Figure C.4: Page from *Kilbarchan Parish Register*, Page 32.



96 *Parish of Kilbarchan.*

Rose, Robert, in Linwood
Elizabeth, 2 Nov. 1655.
James, 22 Aug. 1658.
Rosse, Robert, in Meikle Fultoune
Robert, 12 May 1661.
Ross, Robert, of Kirkland, and Katherine Hamilton, 1688 in Linwood m. 26 April 1677
Grissell, 31 Dec. 1682.
Elizabeth, 9 Dec. 1688.
Agnes, 4 Dec. 1692.
Ross, Robert, and Mary Colquhoun, in Linwood
Christian, 4 June 1697.
Russide, David, in Hill, and Margaret Stuart
Anne, born 7 Sept. 1767.
Sandilands, Thomas, surgeon in Kilbarchan, and Janet Lewis
Margaret, born 9 Feb. 1751.
John, born 8 Mar. 1753.
Sclatter, John, and Jonet Cochran, in Lochwinnoch
Margaret, 4 Feb. 1683.
Sklaitter, Peter, 1655 in Linwood
John, 3 July 1653.
Jonet, 30 May 1655.
Robert, 30 July 1658.
David, 7 April 1661.
Scott, ——, in Plainlees, and Margaret Barbour
——, 31 July 1709.
Scott, Alexander, and Agnes Greive, in Kilmacolm
James, —— June 1683
Scott, Archibald, par. of Largs, and Elizabeth Houstoun, par. in Kirktoun, 1695 in Craigends, 1698 in Kirktoun m. 31 Dec. 1691
Archibald, 24 Feb. 1693.
Francis, 4 Aug. 1695.
Catherine and Anna, 15 Nov. 1698.
John, 3 Jan. 1701.
Scott, John, in Kilbarchan, and Isabella Cumming, par. of Lochwinnoch p. 29 July 1749
Archibald, 17 June 1750.
Janet, born 23 Oct. 1752.
Scott, John, and Janet Wilson p. 19 April 1766
Semple or Sempill, Andrew, and Margaret Waterstoune m. 24 Feb. 1654
Semple, Andrew, in Craigens
Jane, 31 Dec. 1655.
Semple, Andrew, at Mill of Cart
James, 28 Dec. 1656.
Semple, Andro, in Erskinefauld
Elizabeth, 21 April 1661.
Semple, Francis, yr., of Beltrees, and Jean Campbell, par. of Lochgilphead m. Lochgilphead 3 April 1655
Robert, 11 April 1656.
James, 10 May 1657.
Semple, Hugh, in the Kirktoune
Thomas, 2 Nov. 1651.
Robert, 16 July 1653.
Hew, 18 Mar. 1655.
Semple, Hew, in Boghouse
William, 3 April 1657.
Semple, Hugh, of Waterstoune
James, 8 Mar. 1660.
Marie, 20 July 1662.
Sempill, Hugh, in Mossend, and Elizabeth Hall
Hugh, 17 June 1705.

Figure C.5: Page from *Kilbarchan Parish Register*, Page 96.

References

[1] Brad Adelberg. NoDoSE — a tool for semi-automatically extracting structured and semistructured data from text documents. *ACM SIGMOD Record*, 27:283–294, 1998.

[2] Edward H. Adelson, Charles H. Anderson, James R. Bergen, Peter J. Burt, and Joan M. Ogden. Pyramid methods in image processing. *RCA engineer*, 29(6):33–41, 1984. URL `https://alliance.seas.upenn.edu/~cis581/wiki/Lectures/Pyramid.pdf`.

[3] Pieter W. Adriaans and Ceriel Jacobs. Using MDL for grammar induction. *Lecture Notes in Computer Science*, 4201:293–306, 2006.

[4] Pieter W. Adriaans and Paul Vitanyi. The power and perils of MDL. In *IEEE International Symposium on Information Theory, 2007. ISIT 2007*, pages 2216–2220, 2007. doi: 10.1109/ISIT.2007.4557549.

[5] Naveen Ashish and Craig A. Knoblock. Semi-automatic wrapper generation for internet information sources. In *Proceedings of the Second IFCIS International Conference on Cooperative Information Systems, 1997. COOPIS '97*, pages 160–169, 1997.

[6] Moses S. Beach, William Ely, and G. B. Vanderpoel. *The Ely Ancestry*. The Calumet Press, New York, New York, USA, 1902.

[7] Abdel Belad. Retrospective document conversion: application to the library domain. *International Journal on Document Analysis and Recognition*, 1:125–146, 1998.

[8] Abdel Belad. Recognition of table of contents for electronic library consulting. *International Journal on Document Analysis and Recognition*, 4:35–45, 2001.

[9] Dominique Besagni and Abdel Belad. Citation recognition for scientific publications in digital libraries. In *Proceedings of the First International Workshop on Document Image Analysis for Libraries*, pages 244–252, Palo Alto, California, USA, 2004.

[10] Dominique Besagni, Abdel Belad, and Nelly Benet. A segmentation method for bibliographic references by contextual tagging of fields. In *Proceedings of the Seventh International Conference on Document Analysis and Recognition*, pages 384–388, Edinburgh, Scotland, 2003.

[11] Dorothea Blostein and George Nagy. Asymptotic cost in document conversion. In Christian Viard-Gaudin and Richard Zanibbi, editors, *Document Recognition and Retrieval XIX*, pages 82970N–82970N–9, 2012. doi: 10.1117/12.912161. URL `http://spie.org/Publications/Proceedings/Paper/10.1117/12.912161`.

[12] Vinayak Borkar, Kaustubh Deshmukh, and Sunita Sarawagi. Automatic segmentation of text into structured records. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, pages 175–186, Santa Barbara, California, USA, 2001.

[13] Andrew Carlson and Charles Schafer. Bootstrapping information extraction from semi-structured web pages. In *Proceedings of European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases*, pages 195–210, Antwerp, Belgium, 2008.

[14] Gavin C. Cawley. Baseline methods for active learning. *Journal of Machine Learning Research-Proceedings Track*, 16:47–57, 2011. URL `http://jmlr.org/proceedings/papers/v16/cawley11a/cawley11a.pdf`.

[15] Chia-Hui Chang, Chun-Nan Hsu, and Shao-Cheng Lui. Automatic information extraction from semi-structured web pages by pattern discovery. *Decision Support Systems*, 35:129–147, 2003.

[16] Chia Hui Chang, Mohammed Kayed, M.R. Girgis, and K.F. Shaalan. A survey of web information extraction systems. *IEEE Transactions on Knowledge and Data Engineering*, 18 (10):1411–1428, 2006. ISSN 1041-4347. doi: 10.1109/TKDE.2006.152.

[17] Trevor A. Cohn. *Scaling conditional random fields for natural language processing*. PhD thesis, Citeseer, 2007. URL `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.90.1265&rep=rep1&type=pdf`.

[18] Valter Crescenzi, Giansalvatore Mecca, and Paolo Merialdo. Roadrunner: Towards automatic data extraction from large web sites. In *Proceedings of the International Conference on Very Large Data Bases*, pages 109–118, Rome, Italy, 2001.

[19] Aron Culotta, David Kulp, and Andrew McCallum. Gene prediction with conditional random fields. Technical report, University of Massachusetts - Amherst, 2005. URL `http://works.bepress.com/andrew_mccallum/143`.

[20] Nilesh Dalvi, Ravi Kumar, and Mohamed Soliman. Automatic wrappers for large scale web extraction. *Proceedings of the VLDB Endowment*, 4:219–230, 2010.

[21] Charles Elkan and Keith Noto. Learning classifiers from only positive and unlabeled data. In *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '08, pages 213–220, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-193-4. doi: 10.1145/1401890.1401920. URL `http://doi.acm.org/10.1145/1401890.1401920`.

[22] Hazem Elmeleegy, Jayant Madhavan, and Alon Halevy. Harvesting relational tables from lists on the web. *Proceedings of the VLDB Endowment*, 2:1078–1089, 2009.

[23] David W. Embley, Yuan S. Jiang, and Yiu-Kai Ng. Record-boundary discovery in web documents. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, pages 467–478, Philadelphia, Pennsylvania, USA, 1999.

[24] Anna Lisa Gentile, Ziqi Zhang, Isabelle Augenstein, and Fabio Ciravegna. Unsupervised wrapper induction using linked data. In *Proceedings of the Seventh International Conference on Knowledge Capture*, pages 41–48, New York, NY, USA, 2013. ACM.

[25] Terrance Goan, Nels Benson, and Oren Etzioni. A grammar inference algorithm for the world wide web. In *In Proc. of the AAAI Spring Symposium on Machine Learning in Information Access*. AAAI Press, 1996.

[26] Francis J. Grant, editor. *Index to the Register of Marriages and Baptisms in the Parish of Kilbarchan, 1649–1772*. Scottish Record Society. J. Skinner and Company, Ltd., Edinburgh, Scotland, 1912.

[27] Edward Green and Mukki S. Krishnamoorthy. Model-based analysis of printed tables. *Proceedings of the International Conference on Document Analysis and Recognition (ICDAR)*, 1072:214–217, 1995.

[28] Trond Grenager, Dan Klein, and Christopher D. Manning. Unsupervised learning of field segmentation models for information extraction. In *Proceedings of the Forty-third Annual Meeting on Association for Computational Linguistics*, pages 371–378, Ann Arbor, Michigan, USA, 2005.

[29] Peter Grnwald. A minimum description length approach to grammar inference. In *Connectionist, Statistical, and Symbolic Approaches to Learning for Natural Language Processing*, pages 203–216, London, UK, UK, 1996. Springer-Verlag. ISBN 3-540-60925-3. URL `http://dl.acm.org/citation.cfm?id=646314.688520`.

[30] Thomas Gruber. Toward principles for the design of ontologies used for knowledge sharing, 1993.

[31] Yong Zhen Guo, Kotagiri Ramamohanarao, and Laurence A. F. Park. Error correcting output coding-based conditional random fields for web page prediction. In *Web Intelligence and Intelligent Agent Technology, 2008. WI-IAT'08. IEEE/WIC/ACM International Conference on*, volume 1, pages 743–746. IEEE, 2008. URL `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4740540`.

[32] Rahul Gupta and Sunita Sarawagi. Answering table augmentation queries from unstructured lists on the web. *Proceedings of the VLDB Endowment*, 2:289–300, 2009.

[33] Robbie A. Haertel, Eric K. Ringger, James L. Carroll, and Kevin D. Seppi. Return on investment for active learning. In *Proceedings of the Neural Information Processing Systems Workshop on Cost Sensitive Learning*, 2008.

[34] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2): 100–107, 1968. ISSN 0536-1567. doi: 10.1109/TSSC.1968.300136.

[35] P. Bryan Heidorn and Qin Wei. Automatic metadata extraction from museum specimen labels. In *Proceedings of the 2008 International Conference on Dublin Core and Metadata Applications*, pages 57–68, Berlin, Germany, 2008.

[36] Thomas N. Herzog, Fritz J. Scheuren, and William E. Winkler. Phonetic coding systems for names. In *Data Quality and Record Linkage Techniques*, pages 115–121. Springer, 2007.

[37] Jianying Hu, Ramanujan S. Kashi, Daniel Lopresti, and Gordon T. Wilfong. Evaluating the performance of table processing algorithms. *International Journal on Document Analysis and Recognition*, 4:140–153, 2002. ISSN 1433-2833, 1433-2825. doi: 10.1007/s100320200074. URL `http://www.springerlink.com/content/n0r83rnf62jbhq3f/`.

[38] Weiming Hu, Wei Hu, Nianhua Xie, and S. Maybank. Unsupervised active learning based on hierarchical graph-theoretic clustering. *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics*, 39(5):1147–1161, October 2009. ISSN 1083-4419. doi: 10.1109/TSMCB.2009.2013197.

[39] Matthew Hurst and Tetsuya Nasukawa. Layout and language: Integrating spatial and linguistic knowledge for layout understanding tasks. In *Proceedings of the Eighteenth Conference on Computational Linguistics*, pages 334–338, Saarbrcken, Germany, 2000.

[40] Chunyu Kit. A goodness measure for phrase learning via compression with the MDL principle. In *Proceedings of the ESSLLI Student session*, pages 175–187,

1998. URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.50.3179&rep=rep1&type=pdf.

[41] Nicholas Kushmerick. *Wrapper induction for information extraction*. PhD thesis, University of Washington, Seattle, Washington, USA, 1997.

[42] Daniel X. Le and George R. Thoma. Automatically creating biomedical bibliographic records from printed volumes of old indexes. In *Proceedings of the 9th World Multiconference on Systemics, Cybernetics and Informatics*, pages 267–274, Orlando, Florida, USA, 2005.

[43] Kristina Lerman, Craig Knoblock, and Steven Minton. Automatic data extraction from lists and tables in web sources. In *IJCAI-2001 Workshop on Adaptive Text Extraction and Mining*, volume 98, 2001.

[44] Yanliang Li, Jing Jiang, Hai Leong Chieu, and Kian Ming A. Chai. Extracting relation descriptors with conditional random fields. *Proceedings of the 5th International Joint Conference on Natural Language Processing*, pages 392–400, 2011.

[45] Stephen Marsland. Novelty detection in learning systems. *Neural computing surveys*, 3(2): 157–195, 2003. URL http://seat.massey.ac.nz/personal/s.r.marsland/pubs/ncs.pdf.

[46] Andrew Kachites McCallum. MALLET: A machine learning for language toolkit, 2002. URL http://mallet.cs.umass.edu/.

[47] Thomas L. Packer and David W. Embley. Lessons learned in automatically detecting lists in OCRed historical documents. In *Proceedings of the Family History Technology Workshop*, Salt Lake City, Utah, USA, 2012.

[48] Thomas L. Packer and David W. Embley. Cost effective ontology population with data from lists in OCRed historical documents. In *Proceedings of the 2013 ICDAR Workshop on Historical Document Imaging and Processing*, 2013.

[49] Thomas L. Packer and David W. Embley. Scalable recognition, extraction, and structuring of data from lists in OCRed text using unsupervised active wrapper induction. Technical report, Department of Computer Science, Brigham Young University, Provo, Utah, 2014. URL http://deg.byu.edu/papers. (Submitted to TKDD).

[50] Thomas L. Packer and David W. Embley. Unsupervised training of HMM structure and parameters for OCRed list recognition and ontology population. Technical report, Brigham Young University, Provo, Utah, USA, 2014.

[51] Fuchun Peng and Andrew McCallum. Information extraction from research papers using conditional random fields. *Information Processing & Management*, 42:963–979, 2006.

[52] Burr Settles. Biomedical named entity recognition using conditional random fields and rich feature sets. In *Proceedings of the International Joint Workshop on Natural Language Processing in Biomedicine and Its Applications*, JNLPBA '04, pages 104–107, Stroudsburg, PA, USA, 2004. Association for Computational Linguistics. URL `http://dl.acm.org/citation.cfm?id=1567594.1567618`.

[53] Burr Settles. Active learning. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 6(1):1–114, June 2012. ISSN 1939-4608, 1939-4616. doi: 10.2200/S00429ED1V01Y201207AIM018. URL `http://www.morganclaypool.com/doi/abs/10.2200/S00429ED1V01Y201207AIM018`.

[54] Harvey E. Shaffer. *Shaver/Shafer and Dougherty/Daughery Families also Kiser, Snider and Cottrell, Ferrell, Hively and Lowe Families*. Gateway Press, Inc., Baltimore, MD, 1997.

[55] Ray J. Solomonov. A formal theory of inductive inference. *Information and Control*, 1964.

[56] Andreas Stolcke and Stephen Omohundro. Hidden markov model induction by bayesian model merging. *Advances in Neural Information Processing Systems*, pages 11–18, 1993.

[57] Cui Tao and David W. Embley. Automatic hidden-web table interpretation by sibling page comparison. In *Conceptual Modeling-ER 2007*, pages 566–581. Springer, 2007. URL `http://link.springer.com/chapter/10.1007/978-3-540-75563-0_38`.

[58] Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995. ISSN 0178-4617, 1432-0541. doi: 10.1007/BF01206331. URL `http://link.springer.com/article/10.1007/BF01206331`.

[59] J. Gerard Wolff. The discovery of segments in natural language. *British Journal of Psychology*, 68(1):97–106, 1977.

[60] J. Gerard Wolff. Unsupervised grammar induction in a framework of information compression by multiple alignment, unification and search. *Proceedings of the Workshop and Tutorial on Learning Context-Free Grammars*, 2003.