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

THOMAS L. PACKER, Brigham Young University
DAVID W. EMBLEY, Brigham Young University

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facts from lists in OCRed documents and inserting them into an ontology would contribute to making a variety of historical documents machine searchable, queryable, and linkable.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 propose 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 without 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 than a state-of-the-art statistical sequence labeler.

Categories and Subject Descriptors: I.2.7 [**Artificial Intelligence**]: Natural Language Process—*Language parsing and understanding*; H.3.1 [**Information Storage and Retrieval**]: Content Analysis and Indexing

General Terms: Algorithms, Experimentation, Performance

Additional Key Words and Phrases: information extraction, wrapper induction, unsupervised active learning, grammar induction, OCRed text document, list, ontology population, regular expression generation

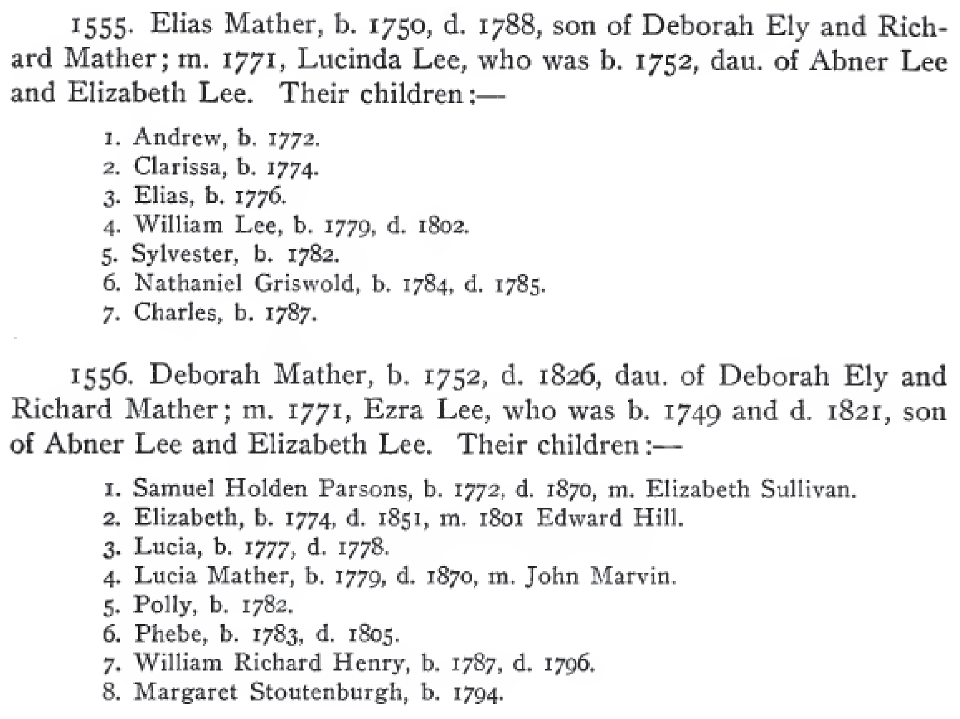
ACM Reference Format:
Thomas L. Packer and David W. Embley, 2014. Scalable Recognition, Extraction, and Structuring of Data from Lists in OCRed Text using Unsupervised Active Wrapper Induction. *ACM Trans. Knowl. Discov. Data.* 0, 0, Article 00 (0000), 47 pages.
DOI:http://dx.doi.org/10.1145/0000000.0000000

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

Authors' addresses: Thomas L. Packer and David W. Embley, Computer Science Department, Brigham Young University.

DOI:http://dx.doi.org/10.1145/0000000.0000000


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.

Fig. 1. Example Text from *The Ely Ancestry*, Page 154

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 [Gruber 1993]. 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. There has been little work 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 1 shows a small part of one family history book, a piece of page 154 of *The Ely Ancestry* [Beach et al. 1902]. It shows two different types of records, members of two lists: parent records and child records.

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 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 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, there is a dearth of formatting cues 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 examples. 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 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 save a lot of 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. 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 2, we survey the previous work most closely related to ListReader. In Section 3, we give an overview of ListReader wrapper induction and execution from a user's perspective. In Section 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 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 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 8, we give current limitations of ListReader, future work, and conclusions.

2. RELATED WORK

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

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 [Wolff 2003], [Wolff 1977] applies a combination of the minimum length encoding (MLE) criterion from information theory, multiple string alignment, and search to perform unsupervised grammar induction. Kit [Kit 1998] 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 [Adriaans and Vitanyi 2007], [Adriaans and Jacobs 2006]. 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) [Goan et al. 1996] and hidden Markov model (HMM) grammars [Stolcke and Omohundro 1993]. 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 [Goan et al. 1996].

2.2. Web Wrapper Induction

Wrapper induction [Kushmerick 1997] 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 [Tao and Embley 2007] 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 [Embley et al. 1999] 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 [Kushmerick 1997], [Ashish and Knoblock 1997], xpaths [Dalvi et al. 2010], finite state automata [Lerman et al. 2001], and conditional random fields [Elmeleegy et al. 2009], [Gupta and Sarawagi 2009]. 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 [Chang et al. 2003]. 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. There appears to be more variability in the field content of OCRed lists than in 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.

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 [Belaïd 1998], [Belaïd 2001] and Besagni, et al. [Besagni and Belaïd 2004], [Besagni et al. 2003] extract records and fields from lists of citations, but rely heavily on hand-crafted knowledge that is specific to bibliographies. Adelberg [Adelberg 1998] and Heidorn and Wei [Heidorn and Wei 2008] 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 proposal 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).

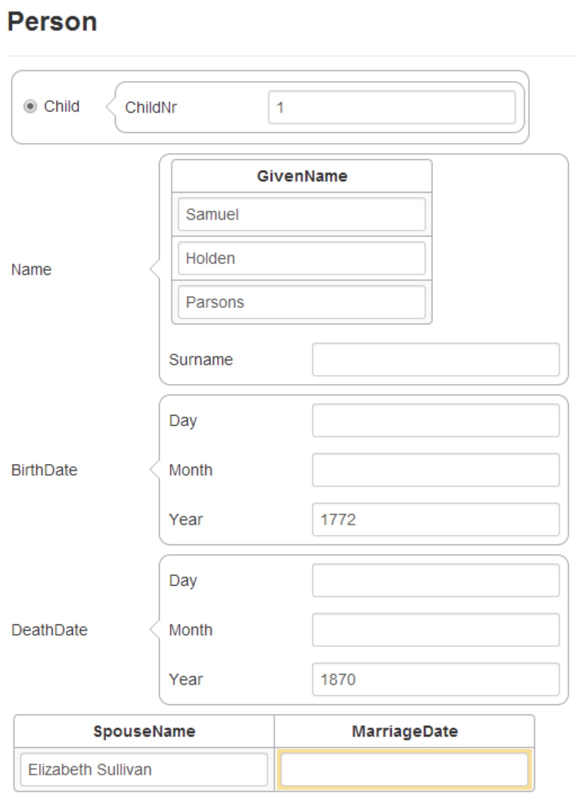
We conclude this section by comparing the ListReader approach described here with our own previous work in this area. In [Packer and Embley 2013] we present both Regex- and HMM-based wrapper induction techniques. While both work well at extracting information from some lists, they have limitations. Both 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 1. The wrapper induction approach described in the remainder of this paper overcomes all of the above limitations.

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 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 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.

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



Person

(●) Child — ChildNr: 1

Name
- GivenName: Samuel; Holden; Parsons
- Surname:

BirthDate
- Day:
- Month:
- Year: 1772

DeathDate
- Day:
- Month:
- Year: 1870

SpouseName	MarriageDate
Elizabeth Sullivan	

Fig. 2. Filled-in Form for Samuel Holden Parsons Record

Figure 2), a multiple-entry form field to accept multiple entries (e.g. the *GivenName* field in Figure 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 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 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 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 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 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 1 with the form in Figure 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 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 2, for example, ListReader creates the diagram in Figure 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 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 3 we can have *Person-SpouseName-MarriageDate*($Person_2$, "Edward Hill", "1801") for the second child, Elizabeth, in Deborah Mather's family in Figure 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 3); and (5) concept categorization hierarchies, including, 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.

ListReader can also use the information obtained from the filled-in form to label the fields within the text as Figure 4 shows. Label names correspond to fields in the form.

[1]Note that the highlighted *MarriageDate* field in Figure 2 is the field of focus awaiting a click on a marriage date, but none is given, so U leaves the field blank.

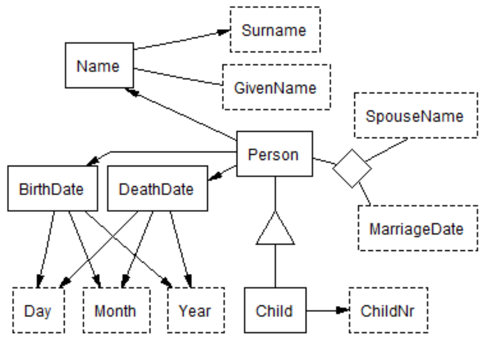


Fig. 3. Ontology Specifying Information of Interest

```
<rkblue</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)>.
```

Fig. 4. Labeled Samuel Holden Parsons Record

Table I. 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”)	

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 I shows how ListReader populates the ontology schema in Figure 3 for the labeled text in Figure 4—thirteen unary predicates, nine binary predicates, and one ternary predicate. Since there is no marriage date, the ternary predicate includes a marked null ($\perp_1$) as a placeholder.

4. REPRESENTATION CORRESPONDENCES

To automate much of ListReader processing, we establish mappings among three types of knowledge representation: (1) HTML forms (e.g. Figure 2), (2) ontology structure (e.g. Figure 3), and (3) in-line labeled text (e.g. Figure 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.

Definition 4.1 (*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 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.

Definition 4.2 (*Fact*). A *fact* is an instantiated, first-order, n-ary ($n \geq 1$) predicate, asserted to be true.

Definition 4.3 (*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 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 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 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 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 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 I, 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 object set to range object set). In Figure 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.

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

Definition 4.4 (*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 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*.

Definition 4.5 (*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 there exists at least one non-cyclic path p from the root object set r to s such that all object sets in p are non-lexical (except for s, itself).

The ontology in Figure 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*>.

Definition 4.6 (*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 2 corresponds precisely to the list ontology in Figure 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 I gives the mapping for the data instances in the filled-in form in Figure 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 2, we can label the first child record of Deborah Mather in Figure 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 shows. Observe that the labels in Figure 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, *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.

Definition 4.7 (*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.

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 [Hu et al. 2009] 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 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.

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* [Shaffer 1997]. 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.

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 [Adelson et al. 1984] and phonetic algorithms like Soundex in searching for historical variants of names [Herzog et al. 2007]. 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-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

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

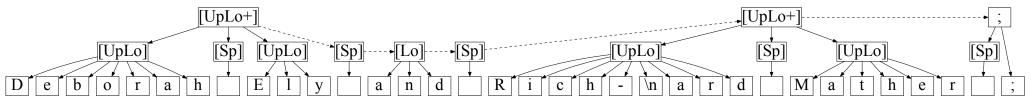


Fig. 5. Conflation Tree of Text "`Deborah Ely and Rich-\nard Mather ;`"

(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 ("` ,`"), semicolon ("` ;`"), 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 5 and 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 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 6, connected by dashed, right-pointing arrows.

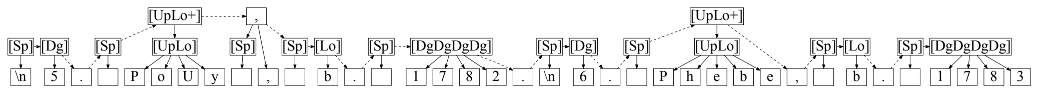

[UpLo+]
,
[UpLo+]
[Sp] [Dg] [Sp] [UpLo] [Sp] [Sp] [Lo] [Sp] [DgDgDgDg] [Sp] [Dg] [Sp] [UpLo] [Sp] [Lo] [Sp] [DgDgDgDg]
\n 5 . P o U y , b . 1 7 8 2 . \n 6 . P h e b e , b . 1 7 8 3

Fig. 6. Conflation Tree of Text "`\n5. PoUy , b. 1782.\n6. Phebe, b. 1783`"

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.

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.

Definition 5.1 (*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 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 from the root 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. There are, therefore, two leaf nodes beneath 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.

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 [Ukkonen 1995] 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

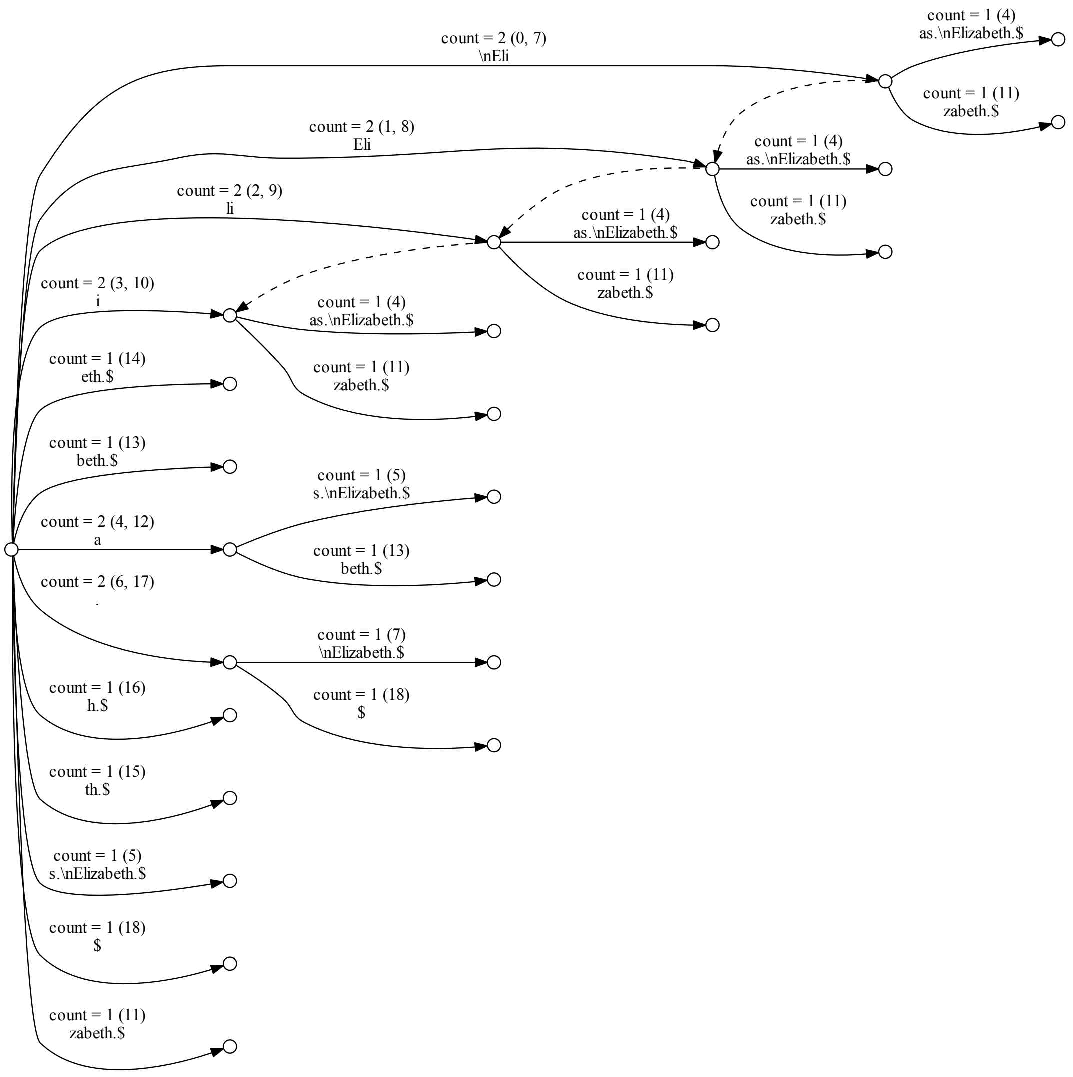


Fig. 7. Suffix Tree of Text “`\nElias.\nElizabeth.`” (plus dashed-arrow back pointers)

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 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 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 1. Observe that each record has the pattern "`[Sp] [Dg] . [Sp] [UpLo+] , [Sp] [Lo] . [Sp] [DgDgDgDg] . [Sp]`". In Figure 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.

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 [Kit 1998]. 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 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

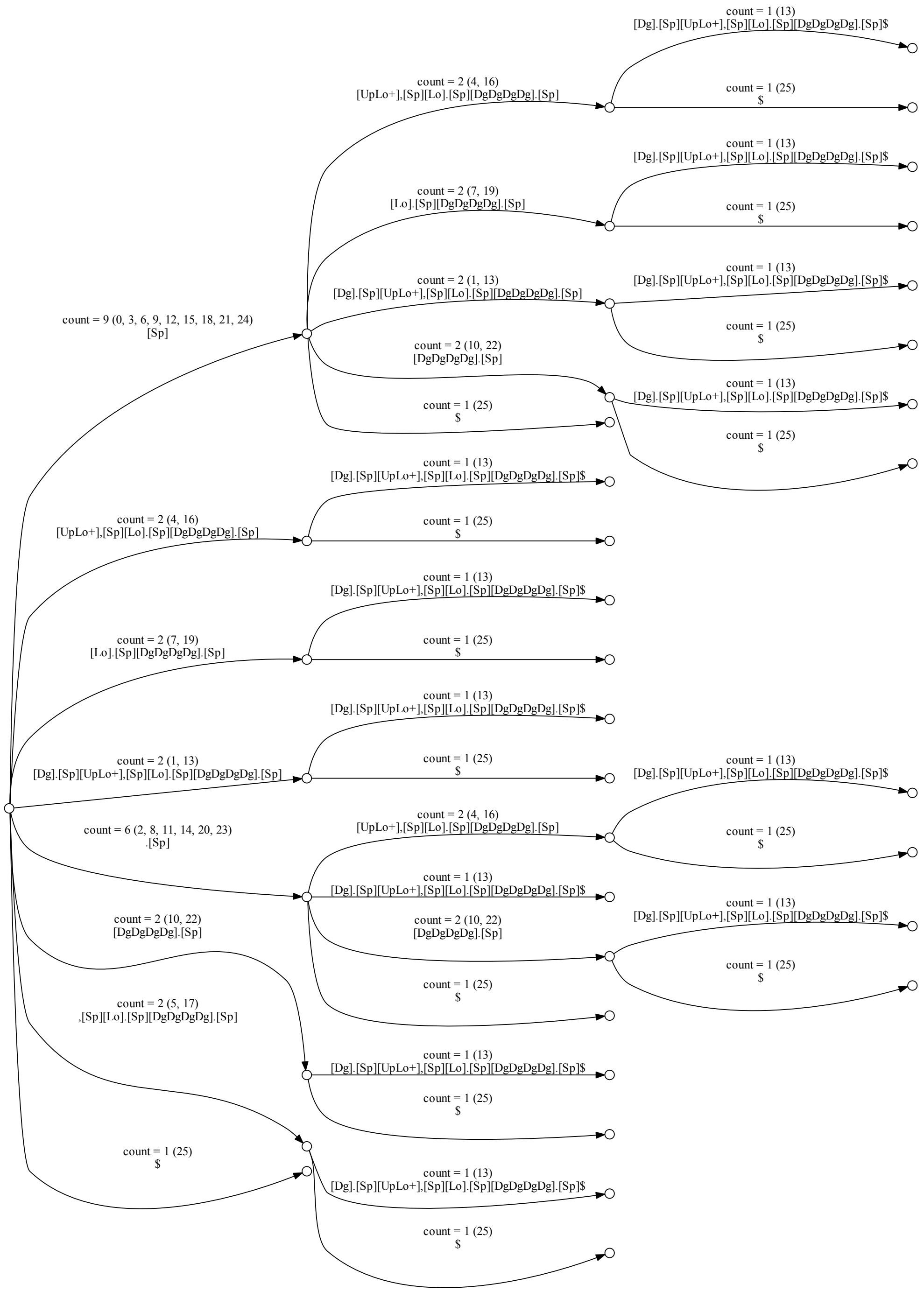

count = 9 (0, 3, 6, 9, 12, 15, 18, 21, 24)
[Sp]
count = 2 (4, 16)
[UpLo+],[Sp][Lo].[Sp][DgDgDgDg].[Sp]
count = 2 (7, 19)
[Lo].[Sp][DgDgDgDg].[Sp]
count = 2 (1, 13)
[Dg].[Sp][UpLo+],[Sp][Lo].[Sp][DgDgDgDg].[Sp]
count = 6 (2, 8, 11, 14, 20, 23)
.[Sp]
count = 2 (10, 22)
[DgDgDgDg].[Sp]
count = 2 (5, 17)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

Fig. 8. Suffix Tree of Conflated Text of the First Two Child Records in Figure 1: “`[Sp][Dg].[Sp][UpLo+],[Sp][Lo].[Sp][DgDgDgDg].[Sp][Dg].[Sp][UpLo+],[Sp][Lo].[Sp][DgDgDgDg].[Sp]`”

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 there should be some minimum. In Figure 1 the single-name, birth-date-only record of focus in Figure 8 repeats six times in Figure 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 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 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 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 9 is the actual cluster of records ListReader would produce from processing the text in Figure 1. Observe that the record pattern is the discovered record pattern in Figure 8. If ListReader creates a suffix tree for the entire text in Figure 1, instead of just the first two child records, it would also produce the second record cluster in Figure 9. With a larger input text, ListReader could potentially identify all six record types present in Figure 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 10 shows for the first record in Figure 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 8.

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).

```
— [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"
```

Fig. 9. Record Clusters from Figure 1

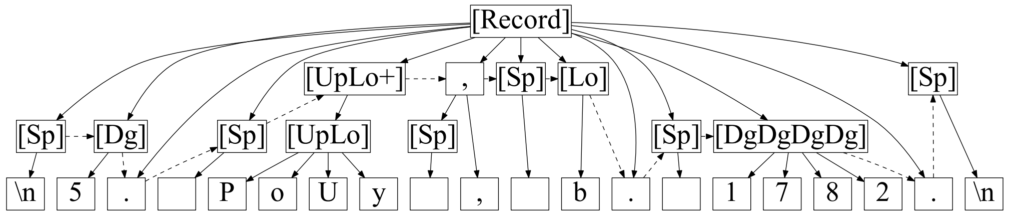


Fig. 10. Phase-one Parse Tree of the Text "\n5. PoUy , b. 1782.\n"

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 1. Observe in Figure 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 1 would also have been

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

Fig. 11. Record Cluster for Two-Child Records

included. The record cluster selected from the longer pattern is in Figure 11, whereas the record cluster selected from the shorter pattern is the first cluster in Figure 9. Now, observe that the substring "2. Clarissa, b. 1774." appears in both clusters and also twice in the cluster in Figure 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.

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 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 11. Thus, in this case, ListReader assigns all three candidate child-record substrings in Figure 11 to the first cluster in Figure 9 and discards the two-record cluster. This is good for two reasons. The records end up in the intuitively best cluster and there is no need to 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.

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 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 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 9, therefore, the identified

[4]Work on information compression justifies our choice of scoring and ranking [Solomonov 1964]. See Appendix B for a full explanation.

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 record 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.

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 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.

5.8. Field Group Parsing

This step extends the conflation parsing that began in Step 2 (Subsection 5.2). Unlike that step, the patterns being matched are the field group templates created in Step 7 (Subsection 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.

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 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 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.

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 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 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 13 shows the parse tree constructed in phase-two from the text of the first record in Figure 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

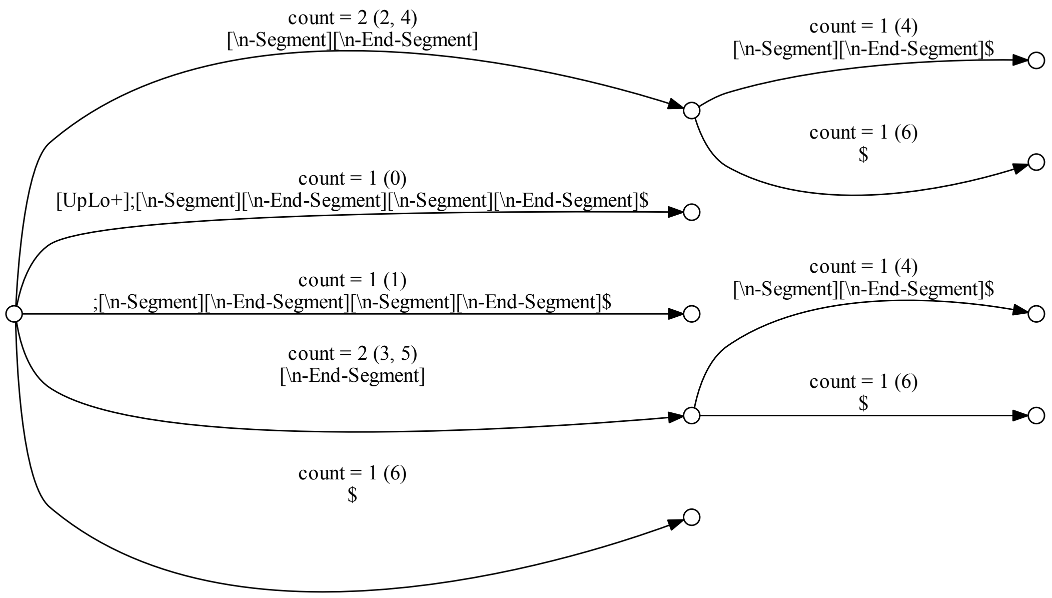

count = 2 (2, 4)
[\n-Segment][\n-End-Segment]
count = 1 (4)
[\n-Segment][\n-End-Segment]$
count = 1 (6)
$
count = 1 (0)
[UpLo+];[\n-Segment][\n-End-Segment][\n-Segment][\n-End-Segment]$
count = 1 (1)
;[\n-Segment][\n-End-Segment][\n-Segment][\n-End-Segment]$
count = 1 (4)
[\n-Segment][\n-End-Segment]$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

Fig. 12. Suffix Tree of "`[UpLo+] ; [\n-Segment] [\n-End-Segment] [\n-Segment] [\n-End-Segment]`"

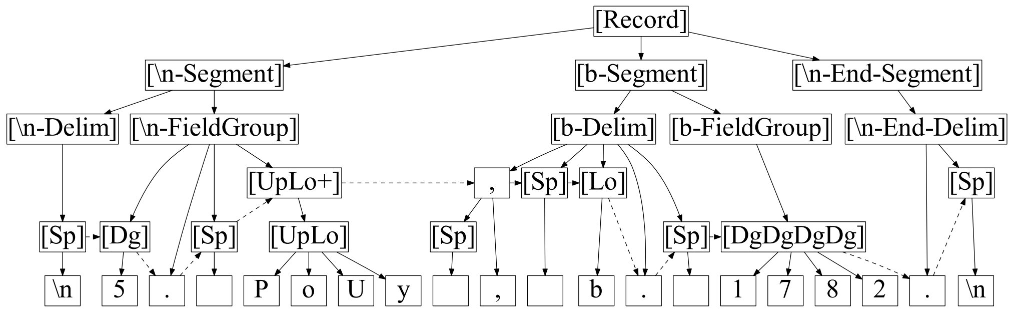


Fig. 13. Phase-two Parse Tree of the Text "`\n5. PoUy , b. 1782.\n`"

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. There are usually much fewer nodes to consider in this step than in phase one. Still, this step also contributes $O(t)$ time and space to the pipeline.

```
([\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])([a-z]+)(\.)([ \n])([\d]{4}))((\.)([\n]))
```

(b)

Fig. 14. Regular Expressions for the Parse Trees in (a) Figure 10 and (b) Figure 13

5.11. Regex Construction

ListReader creates a regular expression from the set of induced parse trees for each record cluster (e.g. Figures 10 and 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 14 gives two regular expressions, which we use as examples as we describe regex construction: (a) for the phase-one parse tree in Figure 10 and (b) for the phase-two parse tree in Figure 13.

The smallest parenthesis-enclosed expressions are for the word-level conflation symbols (e.g. "`[Sp]`", "`[Lo]`", "`[UpLo]`", "`[Dg]`", and "`[DgDgDgDg]`"). Notice in Figure 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 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 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 cleverly 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. There are at most $O(t)$ alternatives for each field group segment. 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.

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 3 and 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.

In most approaches to active learning, there are two key steps: active sampling and model update, with the active sampling step being the hallmark of active learning

[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.

[Settles 2012]. 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 ([Hu et al. 2009]) 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. ([Hu et al. 2009]) propose that there are 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 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 there is more than one such string 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 [Haertel et al. 2008]. 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 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 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 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 [Settles 2012], it is not necessary for ListReader to 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 [Marsland 2002] 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.

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, and inserts them into the ontology. Any remaining unlabeled text produces no output. Given the two record clusters of Figure 9 and the ontology of Figure 3, if the user provides the 9 field labels that would be requested for these clusters after phase-two processing, 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 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.

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.

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.

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* [Beach et al. 1902]. *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 with knowledge of the test data. We selected *Shaver-Dougherty Genealogy* [Shaffer 1997] 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.

[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.
[8]We will make all text and annotations available to others upon request.
[9]Appendix A contains three sample pages.
[10]Appendix A contains three sample pages.

Table II. 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

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 list, 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 (negatives 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 II shows the 46 field labels.

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

Table III. 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)

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 [McCallum 2002].

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 III. 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.

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 [Cawley 2011]. 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.

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. Because the CRF

Table IV.
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

sometimes asks the user to label prose text but ListReader never does, we decided not to count these labelings against the cost for the CRF. This means that the CRF has a slight advantage as it received training data for negative examples (prose text) without affecting its measured cost. During the test phase, the evaluation system measures the accuracy of the extractor only on text for which the systems did not query the user for labels during training or 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) [Cawley 2011]. The rationale is that there does not exist a single, fixed level of cost that everyone will agree is the right 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 IV.

6.4. Results

Figure 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 V 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 V also shows that both versions of ListReader perform better than the CRF in terms 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

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

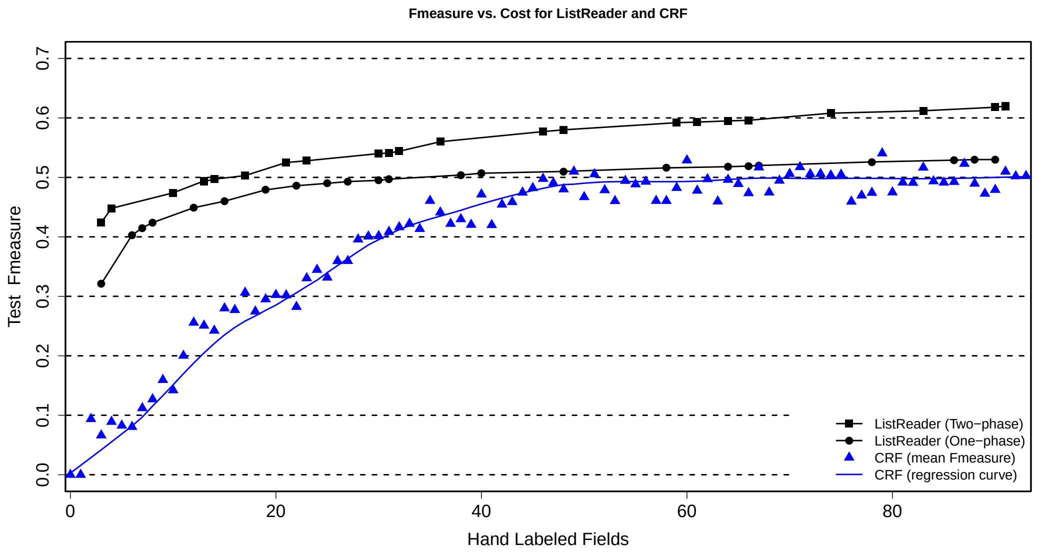


Fig. 15. Labeling/Learning Curves of ListReader and CRF

Table V. 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 Figure 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.

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. Achieving higher precision means there will be less human post-processing 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

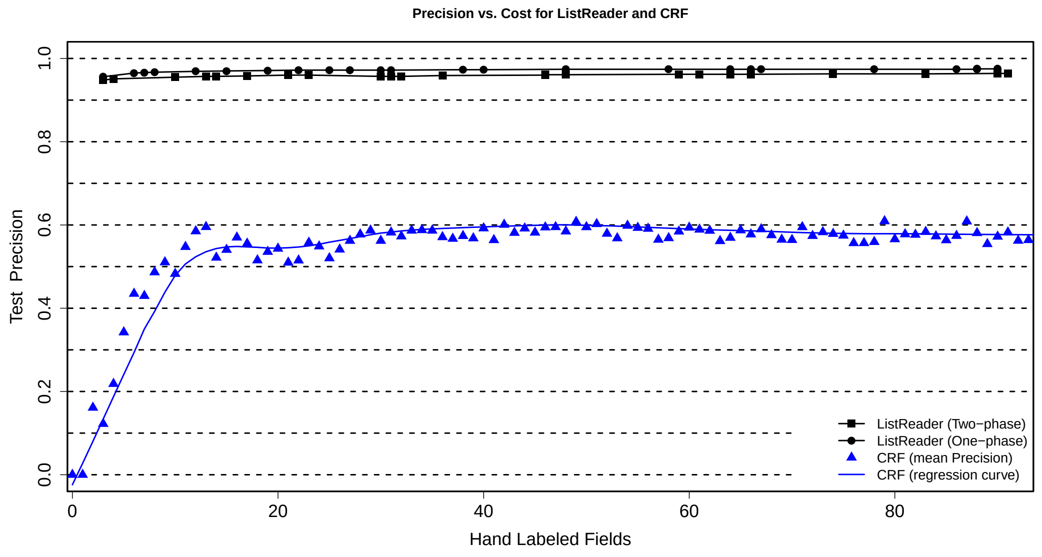


Fig. 16. Labeling/Learning Curves of ListReader and CRF

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 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 5.12), ListReader extracted nearly half of the patterned 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 5) and appears to be better than the CRF's. 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 [Cohn 2007], [Guo et al. 2008]. 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

[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

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.

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 [Packer and Embley 2013], (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 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 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 [Packer and Embley 2013] 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 [Packer and Embley 2013] 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.

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 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

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

of simultaneously improving the accuracy and decreasing the cost of extracting information from OCRed lists. 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.

APPENDIX A: Example Pages

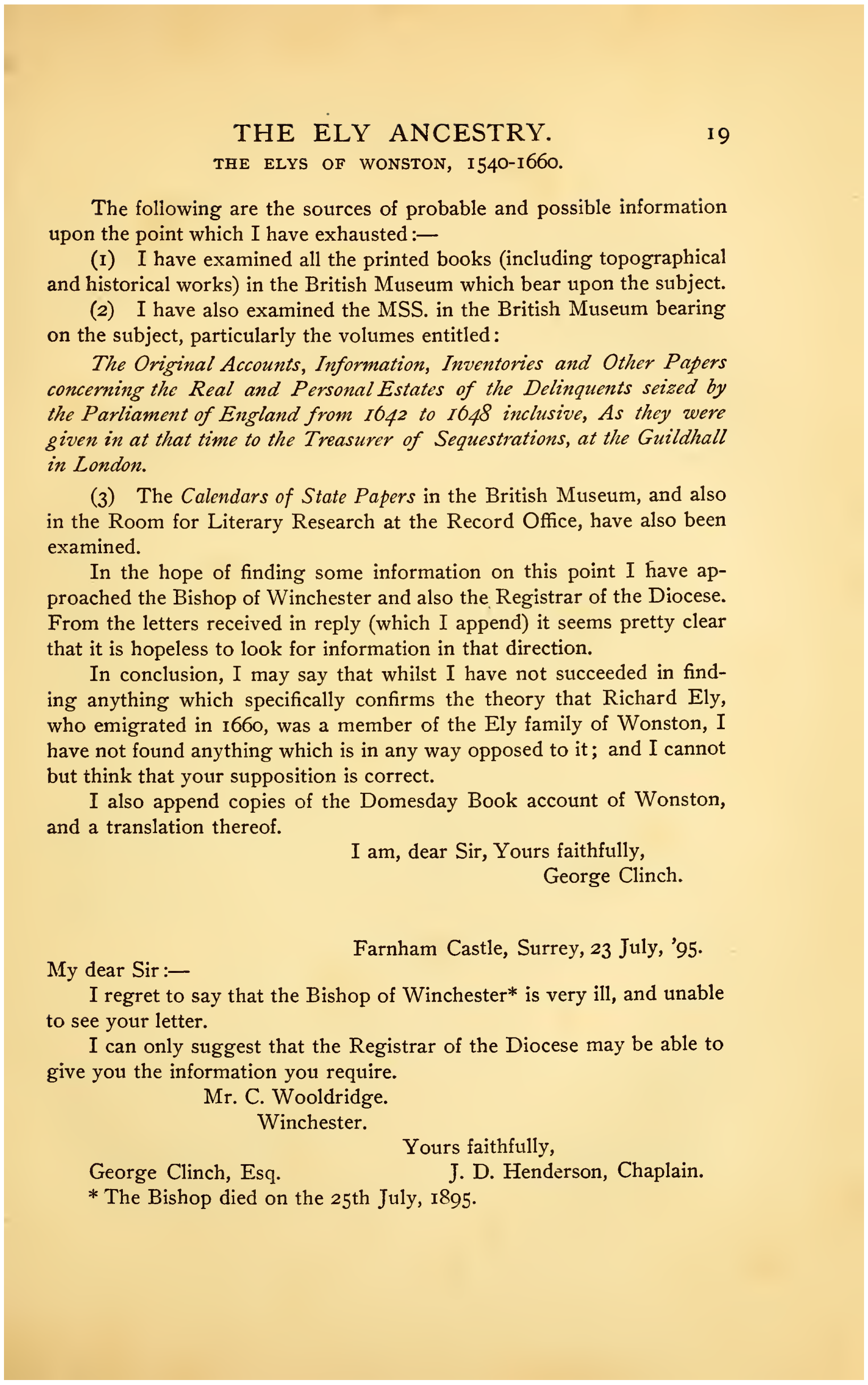
Family history books generally include three kinds of text: (1) Prose/narrative, (2) family lists, and (3) name-index lists (at the back of the book). We have taken examples of these three main types of pages from both of the books discussed in this paper: *The Ely Ancestry* and *Shaver-Dougherty Genealogy*. The examples appear below in Figures 17, 18, 19, 20, 21, 22.

APPENDIX B: Justification of *pattern-length*$\times$*frequency-of-occurrence* for Scoring Clusters

There is 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 a simplification of the usual minimum description length (MDL) formula used in types of grammar induction that are 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 1, we derive the simplified formula ListReader uses to measure the benefit of adding a production rule $G_{lhs} \to G_{rhs}$ 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.)

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

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



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.

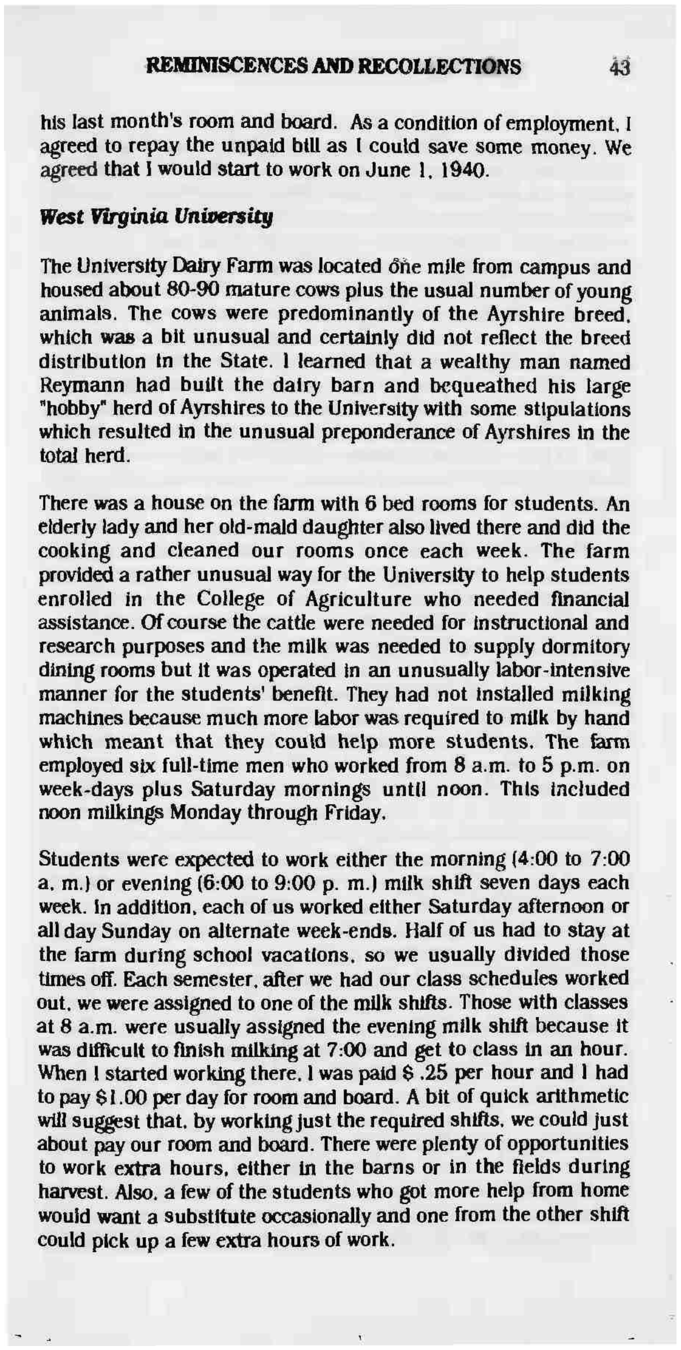
I can only suggest that the Registrar of the Diocese may be able to give you the information you require.

Mr. C. Wooldridge.
Winchester.

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

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

Fig. 17. Prose Page in *The Ely Ancestry*, Page 19



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.

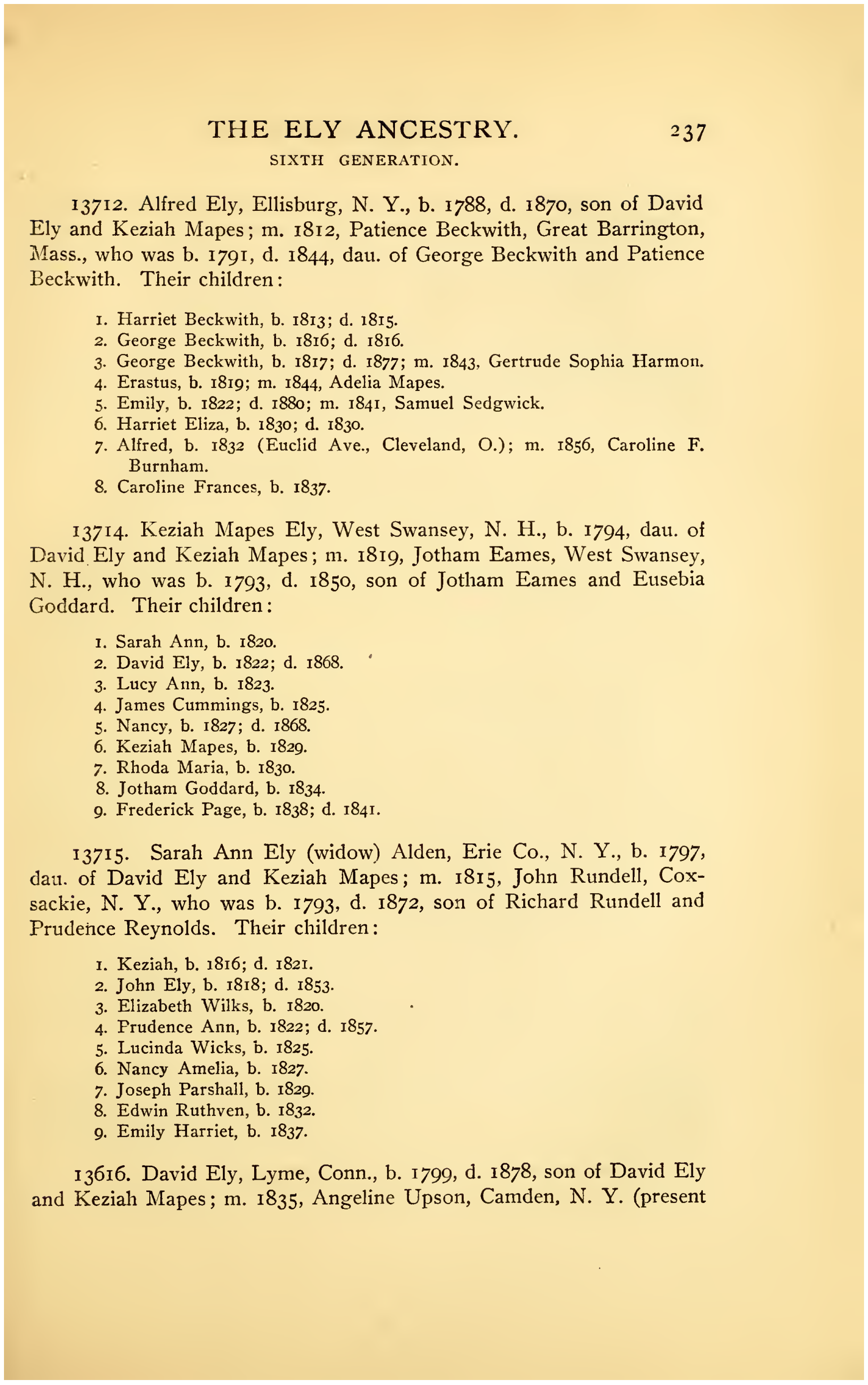
West Virginia University

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.

Fig. 18. Prose Page 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:

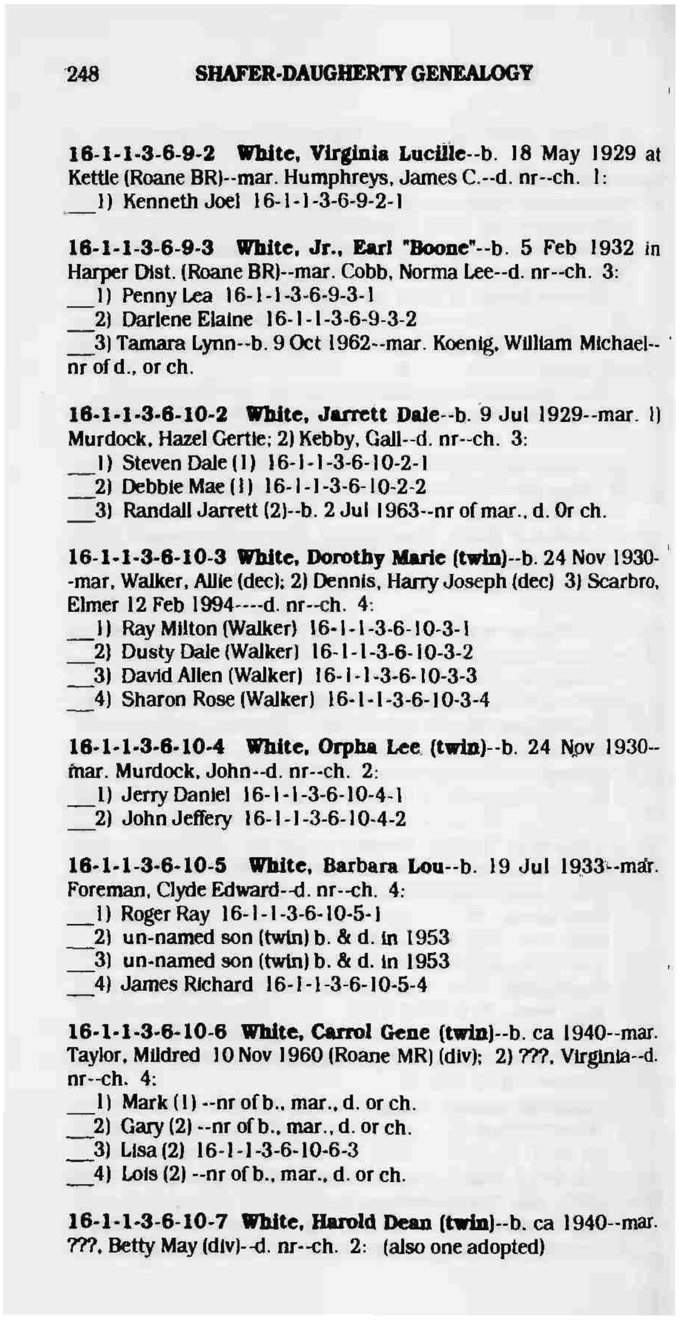
1. Sarah Ann, b. 1820.
2. David Ely, b. 1822; d. 1868.
3. Lucy Ann, b. 1823.
4. James Cummings, b. 1825.
5. Nancy, b. 1827; d. 1868.
6. Keziah Mapes, b. 1829.
7. Rhoda Maria, b. 1830.
8. Jotham Goddard, b. 1834.
9. Frederick Page, b. 1838; d. 1841.

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

Fig. 19. Family List Page 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

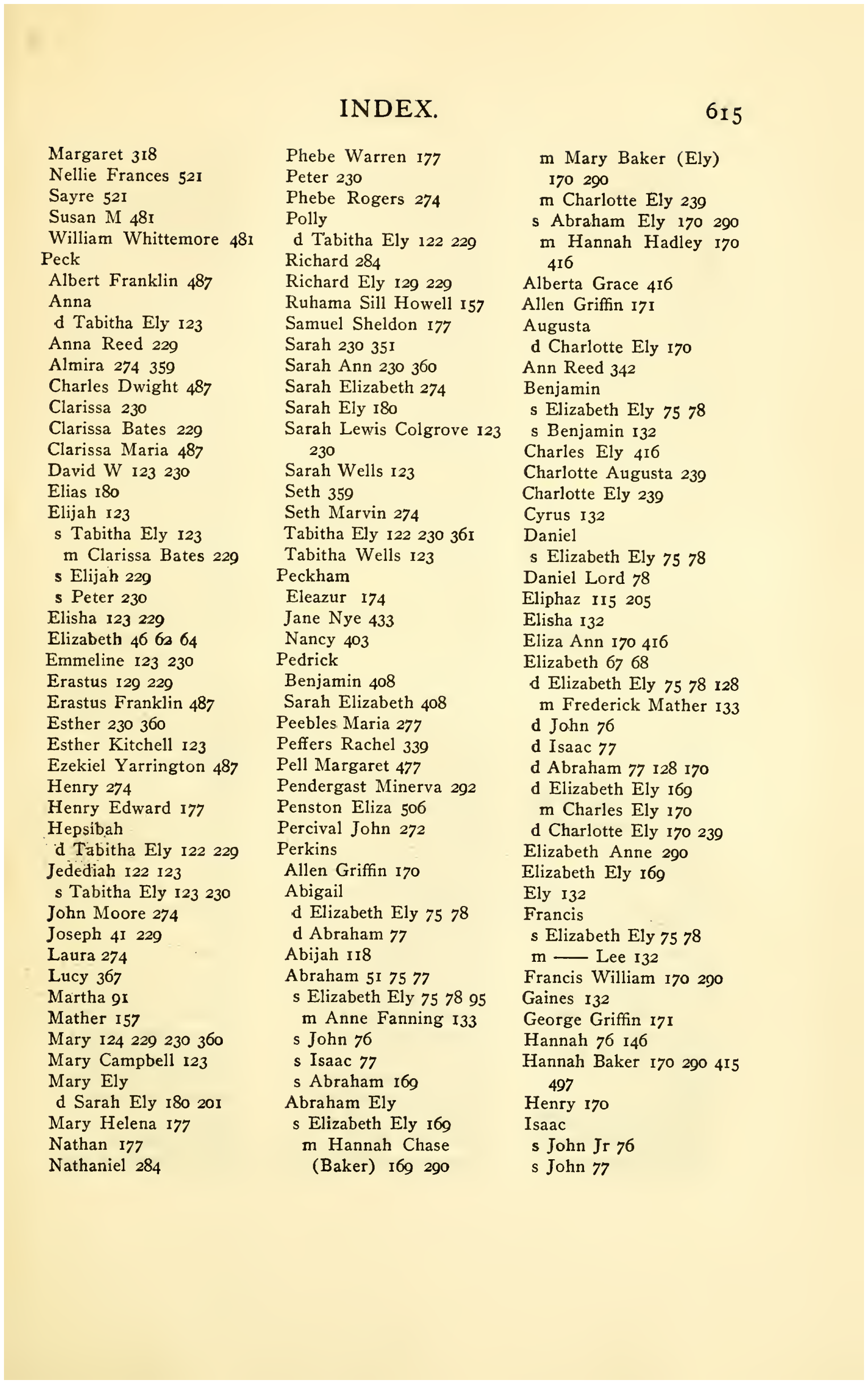
16-1-1-3-6-10-4 White, Orpha Lee (twin)--b. 24 Nov 1930--mar. Murdock, John--d. nr--ch. 2:
___1) Jerry Daniel 16-1-1-3-6-10-4-1
___2) John Jeffery 16-1-1-3-6-10-4-2

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)

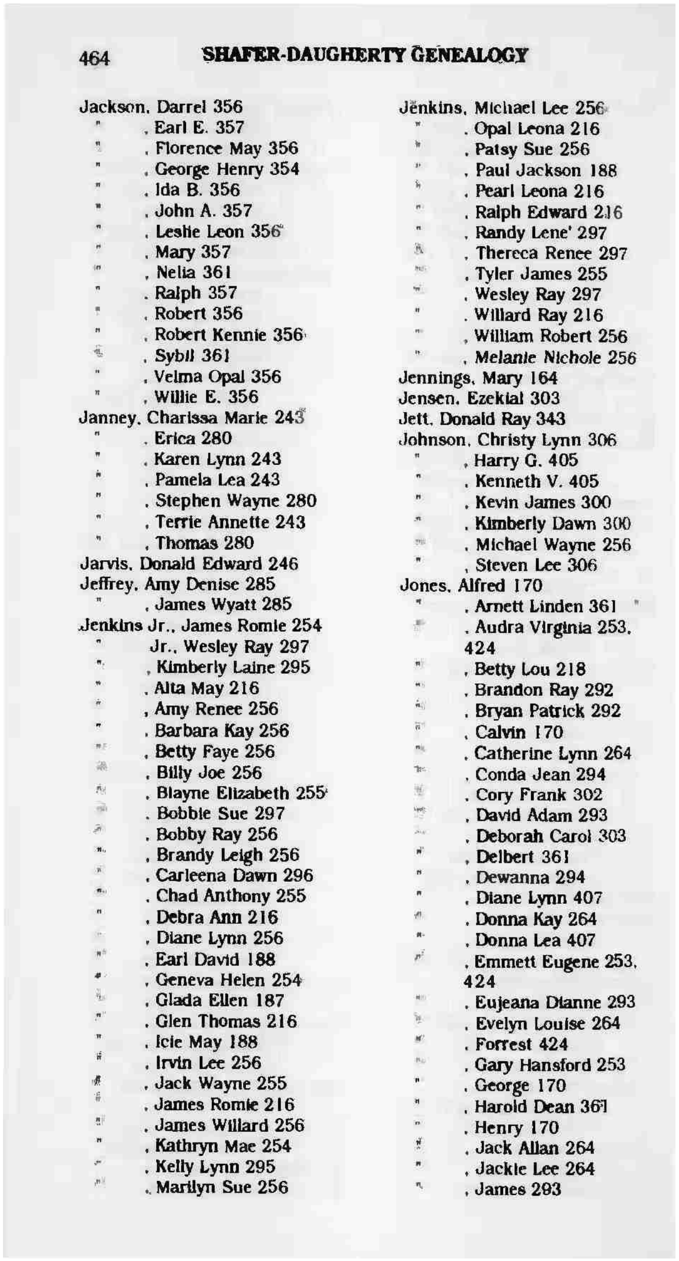
Fig. 20. Family List Page in *Shaver-Dougherty Genealogy*, Page 248



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

Fig. 21. Name Index Page 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

Fig. 22. Name Index Page in *Shaver-Dougherty Genealogy*, Page 464

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 [Solomonov 1964], 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 5.1 guarantees that this threshold is met.

ACKNOWLEDGMENTS

We would like to thank FamilySearch.org for supplying data from its scanned book collection and for their encouragement in this project. We would also like to thank the members of the BYU Data Extraction Research Group, and particularly Stephen W. Liddle, for coding the Annotator used for ground truthing and for interactively supplying labels for ListReader and for their support in supplying additional tools and resources for completing our ListReader project.

REFERENCES

Brad Adelberg. 1998. NoDoSE — A Tool for Semi-automatically Extracting Structured and Semistructured Data from Text Documents. *ACM SIGMOD Record* 27 (1998), 283–294.

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

P. Adriaans and C. Jacobs. 2006. Using MDL for grammar induction. *Lecture Notes in Computer Science* 4201 (2006), 293.

P. Adriaans and Paul Vitanyi. 2007. The Power and Perils of MDL. In *IEEE International Symposium on Information Theory, 2007. ISIT 2007*. 2216–2220. DOI:http://dx.doi.org/10.1109/ISIT.2007.4557549

N. Ashish and C.A. Knoblock. 1997. Semi-automatic wrapper generation for Internet information sources. In *Proceedings of the Second IFCIS International Conference on Cooperative Information Systems, 1997. COOPIS '97*. 160–169.

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

Abdel Belaïd. 1998. Retrospective document conversion: application to the library domain. *International Journal on Document Analysis and Recognition* 1 (1998), 125–146.

Abdel Belaïd. 2001. Recognition of table of contents for electronic library consulting. *International Journal on Document Analysis and Recognition* 4 (2001), 35–45.

Dominique Besagni and Abdel Belaïd. 2004. Citation Recognition for Scientific Publications in Digital Libraries. In *Proceedings of the First International Workshop on Document Image Analysis for Libraries*. Palo Alto, California, USA, 244–252.

Dominique Besagni, Abdel Belaïd, and Nelly Benet. 2003. A Segmentation Method for Bibliographic References by Contextual Tagging of Fields. In *Proceedings of the Seventh International Conference on Document Analysis and Recognition*. Edinburgh, Scotland, 384–388.

Gavin C. Cawley. 2011. Baseline Methods for Active Learning. *Journal of Machine Learning Research-Proceedings Track* 16 (2011), 47–57. http://jmlr.org/proceedings/papers/v16/cawley11a/cawley11a.pdf

Chia-Hui Chang, Chun-Nan Hsu, and Shao-Cheng Lui. 2003. Automatic Information Extraction from Semi-structured Web Pages by Pattern Discovery. *Decision Support Systems* 35 (2003), 129–147.

Trevor A. Cohn. 2007. *Scaling conditional random fields for natural language processing*. Ph.D. Dissertation. Citeseer. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.90.1265&rep=rep1&type=pdf

Nilesh Dalvi, Ravi Kumar, and Mohamed Soliman. 2010. Automatic Wrappers for Large Scale Web Extraction. *Proceedings of the VLDB Endowment* 4 (2010), 219–230.

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

David W. Embley, Y. S. Jiang, and Yiu-Kai Ng. 1999. Record-boundary Discovery in Web Documents. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*. Philadelphia, Pennsylvania, USA, 467–478.

Terrance Goan, Nels Benson, and Oren Etzioni. 1996. 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.

Thomas Gruber. 1993. Toward Principles for the Design of Ontologies Used for Knowledge Sharing. (1993).

Yong Zhen Guo, Kotagiri Ramamohanarao, and Laurence AF Park. 2008. 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*, Vol. 1. IEEE, 743–746. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4740540

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

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

P. Bryan Heidorn and Qin Wei. 2008. Automatic Metadata Extraction from Museum Specimen Labels. In *Proceedings of the 2008 International Conference on Dublin Core and Metadata Applications*. Berlin, Germany, 57–68.

Thomas N. Herzog, Fritz J. Scheuren, and William E. Winkler. 2007. Phonetic Coding Systems for Names. In *Data Quality and Record Linkage Techniques*. Springer, 115–121.

Weiming Hu, Wei Hu, Nianhua Xie, and S. Maybank. 2009. Unsupervised Active Learning Based on Hierarchical Graph-Theoretic Clustering. *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics* 39, 5 (Oct. 2009), 1147–1161. DOI:http://dx.doi.org/10.1109/TSMCB.2009.2013197

Chunyu Kit. 1998. A goodness measure for phrase learning via compression with the MDL principle. In *Proceedings of the ESSLLI Student session*. 175–187. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.50.3179&rep=rep1&type=pdf

Nicholas Kushmerick. 1997. *Wrapper induction for information extraction*. Ph.D. Dissertation. University of Washington, Seattle, Washington, USA.

K. Lerman, C. Knoblock, and S. Minton. 2001. Automatic data extraction from lists and tables in web sources. In *IJCAI-2001 Workshop on Adaptive Text Extraction and Mining*, Vol. 98.

Stephen Marsland. 2002. Novelty Detection in Learning Systems. *Neural Computing surveys* 3, I-39 (2002).

Andrew Kachites McCallum. 2002. MALLET: A Machine Learning for Language Toolkit. (2002). http://mallet.cs.umass.edu/

Thomas L. Packer and David W. Embley. 2013. 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*.

Burr Settles. 2012. Active Learning. *Synthesis Lectures on Artificial Intelligence and Machine Learning* 6, 1 (June 2012), 1–114. DOI:http://dx.doi.org/10.2200/S00429ED1V01Y201207AIM018

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

Ray J. Solomonov. 1964. A Formal Theory of Inductive Inference. *Information and Control* (1964).

Andreas Stolcke and Stephen Omohundro. 1993. Hidden Markov model induction by Bayesian model merging. *Advances in Neural Information Processing Systems* (1993), 11–18.

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

E. Ukkonen. 1995. On-line construction of suffix trees. *Algorithmica* 14, 3 (Sept. 1995), 249–260. DOI:http://dx.doi.org/10.1007/BF01206331

J. G. Wolff. 1977. The discovery of segments in natural language. *British Journal of Psychology* 68, 1 (1977), 97–106. DOI:http://dx.doi.org/10.1111/j.2044-8295.1977.tb01563.x

J Gerard Wolff. 2003. Unsupervised Grammar Induction in a Framework of Information Compression by Multiple Alignment, Unification and Search. *cs/0311045* (Nov. 2003). http://arxiv.org/abs/cs/0311045 Proceedings of the Workshop and Tutorial on Learning Context-Free Grammars (in association with the 14th European Conference on Machine Learning and the 7th European Conference on Principles and Practice of Knowledge Discovery in Databases (ECML/PKDD 2003), September 2003, Cavtat-

Dubrovnik, Croata), editors: C. de la Higuera and P. Adriaans and M. van Zaanen and J. Oncina, pp 113-124.

Received Month 0000; revised Month 0000; accepted Month 0000