Cost Effective Ontology Population with Data from Lists in OCRed Historical Documents

Thomas L. Packer
Brigham Young University
Provo, Utah, USA
tpacker@byu.net

David W. Embley
Brigham Young University
Provo, Utah, USA
embley@cs.byu.edu

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

I. Introduction

Family history books and other machine-printed documents present much of their valuable content in data-rich lists. As one example, the 85,000+ family history books scanned, OCRed, and placed on-line by FamilySearch.org are full of lists containing hundreds of millions of fact assertions about people, places, and events. As an example, Figure 1 shows lists of the children in two families found on page 154 of *The Ely Ancestry* [3]. These lists make many assertions about family relationships and life events. Our goal is to develop a process to extract the diverse kinds of facts from lists in OCRed documents that is robust to OCR errors and relies on as little human effort as possible. In particular, we are concerned with cheaply extracting rich ontological facts from printed lists in which the up-front cost of extracting information is as low as possible.

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

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. Lists in *The Ely Ancestry*, Page 154

We propose ListReader, a robust, general, and cost-effective solution to the challenge of extracting diverse types of facts from lists in OCRed documents. ListReader populates a user-defined ontology with assertions found and labeled automatically. A ListReader user constructs an ontology for a list by building a data-entry form in a custom web interface and fills in the form with the information from the first record of a list. ListReader induces a wrapper and automatically generalizes it to extract asserted information from the remaining records of the list. Only when ListReader encounters a new field in a later record should it ask the user to update the form to accommodate the new field and insert the field value to provide additional training data. This is the minimum amount of effort conceivable as the user begins a new knowledge extraction project in a new domain and document genre, with no previously assembled resources. After ListReader has begun inducing grammars and extracting information from a document, it can switch into a self-supervised mode in which it uses its store of knowledge to effectively label its own training data for other lists, potentially removing the human user from the process. In this paper we focus only on inducing wrappers from scratch—a process we call semi-supervised wrapper induction. We leave self-supervised wrapper induction

for future work.

Wrapper induction is the automated process of constructing a model (i.e. a grammar) that can extract data from a source document and present it in a uniform format [13]. Each induced wrapper is specifically designed for one data source among many, making it potentially more accurate than applying a single, general model to all of the data sources.

Most work in wrapper induction is specialized for machine-generated HTML pages, although there are a few projects that target lists in HTML documents [9], [10], [14]. The focus on HTML input is reflected in these works' choices in wrapper formalism, which include the following: sets of left and right field context expressions [2], [13], xpaths [8], finite state automata [14], and conditional random fields [9], [10]. These formalisms generally rely on consistent landmarks that are not available in OCRed lists for three reasons: OCRed list text is less consistently structured than machine-generated HTML pages, OCRed text does not contain HTML tags, and field delimiters and content in OCRed documents may contain OCR and typographical errors.

Though not commonly identified as wrapper induction, certain research in extracting information from OCRed printed lists fits our definition and targets input that is similar to our own. Most of this work limits itself to certain kinds of input lists and in applying induction processes that are less adaptable and scalable than our proposal. Belaïd [4], [5] and Besagni, et al. [6], [7] extract records and fields from lists of citations, but rely heavily on hand-crafted knowledge that is specific to bibliographies. Adelberg [1] and Heidorn and Wei [12] 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. Also, the extracted information is limited in ontological expressiveness, which is true of all existing work in wrapper induction of which we are aware.

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

II. ListReader

A. Overview

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

First, a user selects an OCRed image (e.g. a two-layer PDF file) that contains a list (e.g. Figure 1), spots a list and, with ListReader's form interface, constructs a form for the data fields in the first record of the list and fills in the form with text from its first record. For example, supposing the spotted list is the second child list in Figure 1, the user would construct the form in Figure 2 and fill it in by clicking on the words in the PDF document for each field in the form.

Second, from the empty form, ListReader creates the schema of an ontology (e.g. Figure 3).

Third, ListReader uses the information obtained from the filled-in form to label the fields within the OCRed text as training data. Figure 4 shows the labeled text for our example.

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

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

B. Automatic Mappings

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). The mappings establish a one-to-one correspondence among nested form elements, paths in an ontology graph, and path expressions in text labels. Consider, for example, the birth year "1772" in the child record for Samuel in Figure 1. In the filled-in HTML form in Figure 2, "1772" is in the *Year* field, which is nested under the *BirthDate* field, which is at the top level of nesting under the form title *Person*. This entry in the form generates object and relationship instances in the ontology structure in Figure 3: an object identifier to represent Samuel in the object set *Person*, which relates to another object identifier to represent Samuel's birth date in the object set *BirthDate*, which relates to the text string "1772" in the lexical object set *Year*. In-line labels on fields in text identify these same paths; thus the label for "1772" in Figure 4 is *Person.BirthDate.Year*.

[1]We have written all the algorithms discussed in this paper from scratch in the Java programming language, using only the libraries typically included in a Java installation, such as the regular expression package for executing a given regular expression on a given text string.

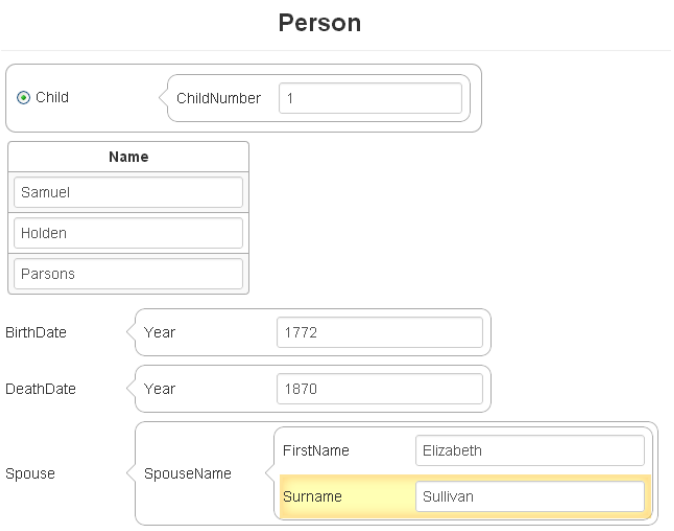

Fig. 2. Filled in Form for Samuel Holden Parsons Record

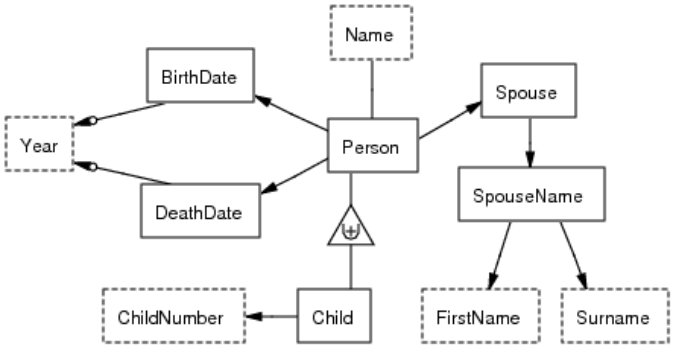

Fig. 3. Initial List Ontology for Samuel Holden Parsons List

Because ListReader marks text with labels that have a one-to-one correspondence with the form and ontology for the list, ListReader's post-processor can fill in the form for the list or, as in our implementation, directly populate the ontology with objects and relationships just as the process that populates the ontology from a manually filled-in form does.

The metaphor of form fill-in for obtaining information is familiar to most users, as is form creation from the basic set of primitives we provide. Our form primitives include the following: a single-entry blank to accept single values (e.g. the birth year "1772" in Figure 2), a multiple-entry blank to accept multiple entries (e.g. in the *Name* field in Figure 2, the three entries "Samuel", "Holden", and "Parsons"), and radio buttons and check boxes to respectively accept one role or several role designations (e.g. the radio button to designate the *Child* role of *Person* in Figure 2). The nesting of form elements provide for relationships among the form elements whose leaf elements are for text objects. Elements with the same name may appear in more than one place in the form, allowing for non-tree-shaped ontologies. The title of the form, *Person* in our example, designates the main object—the object the record describes.

An ontology, rendered as a rooted graph, contains nodes (concepts or sets of objects) and edges (sets of relationships among objects). Nodes may also represent entire concept

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

Fig. 4. Labeled Samuel Holden Parsons Record

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

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

C. Actively Learning Novel Structures

Active learning is a technique within the field of machine learning to reduce the cost of obtaining labeled training data. The learning system, itself, actively identifies a smaller set of examples whose labels would provide greater utility than a randomly selected set. We may distinguish among active learning methods by their query policies, many of which are described in [16]. These sampling policies all assume that the learning system already knows all candidate labels; querying is a matter of reducing the uncertainty about which one of these known labels should be applied to the unlabeled examples.

In our application, we introduce a new assumption to drive active learning, namely that not all labels are known to the

system at the time of a query and that the most helpful query policy is one that is primarily based on novelty detection in that it identifies new structures for which a label is most likely unknown. We are concerned with reaching the minimum possible number of labeled examples which is exactly one label per field. In Sections II-D and II-E we explain how to identify new fields using ListReader's regex and HMM wrappers respectively. The basic idea in both cases is to identify when two records are structurally similar to each other except for the insertion of a new field at some identifiable position in one of the two records.

For example, comparing the first and second records in the second list of Figure 1, ListReader recognizes that the year of Elizabeth's marriage to Edward ("1801") is a new field and asks for its label. When asked for a label for some identified text, a user responds by altering the form for the list, adding a new entry blank for the field, and filling in this text box by clicking on the appropriate string within the PDF. For the marriage-year example here, the user could add a single-entry blank called *MarriageYear* to the end of the form in Figure 2 and click on "1801" in the list in Figure 1 (which in our implemented interface sits side-by-side on the screen with the form). This, in turn, alters the ontology in Figure 3, adding a new object set *MarriageYear* and a new functional relationship set from *Person* to *Person* to *MarriageYear*. It also provides a new path expression *Person.MarriageYear* for labeling "1801". Alternatively, the user could connect the new *MarriageYear* field with the *Spouse* field to create a double-column multiple-entry blank, which would produce a ternary relationship involving the person, the spouse, and the marriage date. The *SpouseName* field in Figure 2 would be nested inside of *Spouse*, the first of the two columns. The user would also need to click on "1801" to copy it into the first slot of the *MarriageDate* column.

D. Adaptive Regex Induction

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

During initialization, ListReader begins learning from nothing more than the text of an OCRed page with the fields of the first record of a list labeled. ListReader initializes a new regex to model the text and labels of the first record using a flat sequence of capture groups. Each capture group corresponds to a field or delimiter. Consider, for example, the following labeling of the first record of the first child list in Figure 1:

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

From this labeling, ListReader generates the initial regular expression (regex) in Figure 5, a first level generalization of the field delimiters and content.

During A* search [11], ListReader generalizes the initial wrapper to produce a set of regexes, one for each record type. That is, ListReader performs an A* graph search once for each line of text below the first record. (ListReader conducts no search if a known regex already matches a record.) Each search traverses a hypothesis space whose nodes are regexes and whose edges are edit operations transforming one regex into another. ListReader uses the first record of the list as the start state in all searches within the same list. Goal states

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

Fig. 5. Regex Induction for First Child List in Fig. 1

are defined as any regex that matches the entire text of an unlabeled record—the record on which search is performed. The purpose of the search is to find the goal regex with the shortest edit distance from the initial regex.

To generate one regex from another, ListReader applies one of four operators to one capture group position. The operators are insertion, deletion, character class expansion, and word length expansion. The insertion operator inserts a one-word capture group of high generality, e.g. "\S{1,10}", with "Unknown" as its field label. The deletion operator deletes a capture group. The two expansion operators allow a child regex to match a larger class of text than its parent. The character class expansion operator, when applied to a field capture group, moves its text up a shallow hierarchy of character classes, e.g. changing "\w{6,6}" to "\S{6,6}". The character class expansion operator, when applied to a delimiter capture group, replaces each character in its text with a predefined set of common OCR error substitutions, e.g., it replaces "[.]" with "[.,]". The word length expansion operator is not used for delimiters, but for fields it expands the upper and lower length bounds of a word by a predetermined increment, e.g., it replaces "\w{6,6}" with "\w{4,9}".

To control the enormously large search space—millions of states for our small example and hundreds of billions of states for larger ones in our test set—we use an A* search strategy with a carefully designed admissible heuristic. A ListReader A* search iterates over a priority queue of regular expressions. ListReader initializes the queue with a regex based on the initial labeled record (e.g. the Initial Regex in Figure 5). On each iteration, the highest-priority regex, r, is dequeued and expanded, meaning that adjacent regexes are generated and added to the queue. The priority[2] of r is $f(r) = g(r) + h(r)$. The $g(r)$ term is the known edit distance from the initial regex to r and the $h(r)$ term is an admissible heuristic estimate of r's remaining distance to a goal. A*'s use of $f(r)$ results in searching paths with the lowest estimated total length. To be admissible, $h(r)$ must never over-estimate the true distance to the nearest goal, ensuring a valuable property of A* search as a whole: it is guaranteed to find the closest goal state first. This makes the search stopping criteria straightforward and the search optimal given the heuristic function. Edit distance is the sum of the edit costs of each operator used to convert

[2]Low values have high priority

one regex into the other. We set the base cost of all operators to 1.0, and we add 0.1 for deletions and insertions that do not occur immediately adjacent to other deletions and insertions, respectively.

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

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

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

The first term, $order_o(r)$, is infinite if o was applied to r "out of order", and zero otherwise. The same set of operators will ultimately produce the same goal states regardless of their order of application if those operator instances can be applied independently of each other. To generate the RecordType2 regex from the Initial Regex in Figure 5, for example, ListReader could have inserted new capture groups after *Name* and after *BirthDate.Year* in either order. Such arbitrariness produces a much larger branching factor for a search space than is necessary for finding the goal state. The $order_o(r)$ term imposes a single left-to-right order of all operator instances that can be applied independently of each other.

The second term, $match_{o,t}(r)$, is infinite if o targets a *hitting minimal constituent* in r, and zero otherwise. A *constituent* c is any contiguous subsequence of one or more capture groups within r. A constituent *hits* when it matches at least one substring of t. A constituent *misses* if it fails to hit. A *minimal constituent* is any constituent in r of length l, where l is the size of the smallest constituent in r that misses. The $match_{o,t}(r)$ term restricts applications to constituents that miss, testing constituents against t from small to large until we find a minimal missing constituent. Minimal constituents that already hit need not be targeted as a location of an operator until we attempt to resolve neighboring missing constituents. For example, the *Name* field in the InitialRegex in Figure 5 misses when applied to the fourth record in Figure 1 because its length must be generalized. Also, the constituent composed of that *Name* field plus the following field delimiter also misses, because a second *Name* field must be inserted between them. However, ListReader cannot recognize the need of the second edit (without extra trial and error) until each of its component capture groups has become generalized enough to hit. The $match_{o,t}(r)$ term provides the most specific location information possible about where the edits might need to happen next to make progress toward a goal state.

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

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

After producing a Regex wrapper (e.g. the *Final Regex* in Figure 5 as a disjunction of *RecordType1* and *RecordType2*), ListReader executes each record type regex against the unlabeled text, removing segments of text as they match. ListReader uses the labels applied by the regex wrapper to the field text of each record to instantiate the ontology for the list by creating objects and relationships as described in Section II-B.

E. Adaptive HMM Induction

An HMM:

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

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

For ListReader, we begin by modeling each word in the text as a member of Y using a multinomial distribution and each field label as a member of S. During initialization, ListReader collects word and label co-occurrence statistics from the first hand-labeled record and trains the HMM using MLE except for modifications to make the most of sparse data and appropriately model the structure of list text. Figure 6 shows an HMM initialized for the first record of the first list of Figure 1. Solid lines represent transition or emission distributions trained with whole single counts from the hand-labeled record, and dotted lines represent transitions trained with fractional counts (non-zero Dirichlet priors). We omit emission model parameters with non-zero priors in this figure for simplicity.

To make the most of our very sparse training data, we employ three techniques. First, we apply a non-zero Dirichlet

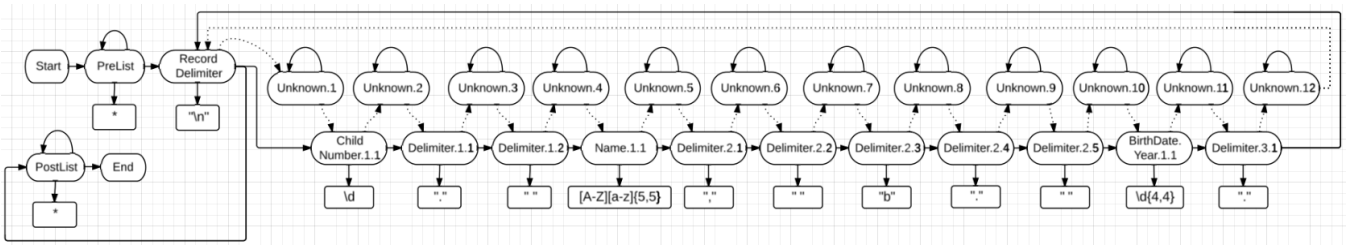

Fig. 6. HMM Initialized for the First Record in Figure 1.

prior to all parameters that should be non-zero using knowledge of how list text behaves. Dirichlet priors are a versatile form of parameter smoothing that amounts to adding fractional pseudo-counts to the training data where prior knowledge dictates. For example, even though we do not see a 10-character-long *Name* in the first record of Figure 1, we know that such a name could exist if we had more training data. We do not want the probability of such a name to be zero. ListReader uses a prior of one divided by the number of words in the list for all words emitted from all states except the *RecordDelimiter* state which can only emit a record delimiter character for the lists we consider.

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

Third, we conflate words from how they naturally appear before performing MLE or applying a trained model to text. To conflate a word, we replace each of its characters with a unique symbol representing which of five character classes it belongs to: uppercase letters, lowercase letters, digits, punctuation, and whitespace. These are represented as regular expressions in Figure 6. Word conflation provides a good balance between discrimination and robustness. Word conflation and smoothing also help allow for OCR and other errors. To avoid drift, we do not further conflate words or adjust word lengths within the emission model beyond the conflated hand-labeled text described here as induction proceeds.

To appropriately model the structure of list text, we modify the HMM in three additional ways. First, we constrain the transition model to have a simple cyclical structure. That is, we avoid transitions from a state to itself and other arbitrary transitions by creating an expanded set of states. This set not only contains separate states for non-list text, record boundaries, field delimiters, and each of the user-specified field labels, but also separate states for individual positions within the word-sequence of each category (hence the numeric state label suffixes in Figure 6). These distinct states allow us to give our HMM a strict linear structure from one record delimiter to the next and prevent erroneous loops and short-cuts through the state graph. To produce a precise cyclical structure, we allow the record delimiter state to transition *from* only the pre-list text state and the last state of a record and to transition *to* only the first state of the record and the post-list text state.

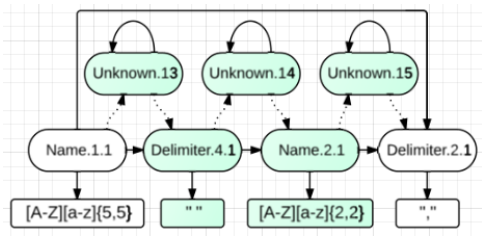

Fig. 7. New HMM Components (Shaded) after Active Learning for Second Name Field.

Second, we loosen the rigid transition model just enough to allow for deletions and insertions. To allow for deletions, we employ parameter smoothing in the transition model, but only for pairs of states obeying the "total order" specified by the training data. For example, we give a non-zero prior to the *Name-BirthDate.Year* transition and a zero prior to the *BirthDate.Year-Name* transition. To allow for insertions, we add unique *Unknown* states between each field or delimiter state in the HMM, giving higher priors to *Unknown* states that sit at key positions, such as the beginning and ending of delimiters, where knowledge of list structure suggests insertions are more likely to occur. For example, consider applying the HMM in Figure 6 to the fourth record of the first list in Figure 1. The text "4. " is exactly what the state sequence *ChildNumber.1.1*, *Delimiter.1.1*, and *Delimiter.1.2* expects. Also, the text ", b. " is exactly what the state sequence *Delimiter.2.1* to *Delimiter.2.5* expects. However, the *Name* state does not expect any of the three intervening tokens "William", " ", or "Lee". Neither of the names are of the expected length and the space is not of the expected character class. However, because of smoothing in the emission models of the intervening states, the Name state will match "William" and the state *Unknown.5* will match " Lee". This is the most likely sequence of matching states because *Unknown.5* appears at the beginning of a long sequence of delimiter states and is therefore given a higher Dirichlet prior. Therefore, ListReader labels " Lee" as *Unknown*.

Third, we model the distinct variability behavior of delimiter and field text in the emission model. Since delimiter text varies less than field text, we prevent ListReader from applying the word conflation described above to any word types found in a known field delimiter. For example, given the delimiter before the birth date in the first record of Figure 1, we prevent the word types ",", " ", "b", and "." from being conflated wherever they appear in the page, leaving them as they are.

During active learning, ListReader queries the user for the labels of each *Unknown*-labeled piece of text. For each query, ListReader trains a new HMM fragment on the newly-labeled text using MLE and the six modifications just described. It

TABLE I. METRICS

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

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

TABLE II. ONTOLOGY POPULATION COST EFFECTIVENESS (%)

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

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

inserts this HMM fragment into the original HMM at the same position as the matching *Unknown* state. The probability of the transition entering a new HMM fragment is split with the original transition bypassing the new fragment. It then executes the whole HMM to identify other *Unknown*-labeled text covered by the new HMM fragment before querying the user again. Figure 7 shows a fragment produced after querying the user about "Lee". After performing active learning on the whole first list of Figure 1, the HMM in Figure 6 is almost twice as long, containing new states for *Name*, *DeathDate.Year*, delimiter text before each, and nine new *Unknown* states. When no *Unknown* text remains, ListReader translates labeled fields to instantiated predicates and inserts them into the ontology as explained in Section II-B.

III. EXPERIMENTAL EVALUATION

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

A key part of ListReader is a machine-learned sequential labeler. The wrapper formalism listed in Section I that should be most capable of modeling text in OCRed lists is the Conditional Random Field (CRF), which is a general approach to sequence labeling, achieving state-of-the-art performance in a number of applications. Therefore, we have chosen to compare ListReader to a CRF [15]. To make our labeling task learnable by the CRF, to reduce over-fitting the training data, and generally to ensure a fair test, we tuned its hyper-parameters and selected an appropriate set of word features on a subset of our data. The features we used were (1) the word text itself, and flags indicating which of the following dictionaries the word appears in: (2) given names, (3) surnames, (4) common words with functional part-of-speech including articles, prepositions, and auxiliary verbs, (5) numerals, and (6) initials (a capital letter followed by a period). We also applied the features of immediate neighbors to each token to provide contextual clues. Our dictionaries are large and have good coverage and constitute a greater amount of knowledge engineering than we allow for ListReader.

As test data, we randomly selected sufficient pages from *The Ely Ancestry* [3] to obtain and isolate the text of 60 child lists. These lists contain 3088 non-space word tokens with 1254 field strings to be identified and extracted. Each list contains between 1 and 12 records for a total of 271 records, with records containing a minimum of 2 fields and a maximum of 12 fields with an average of 4.6 fields per record.

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

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

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

TABLE III. WRAPPER INDUCTION LEARNING ACCURACY (%)

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

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

TABLE IV. ACTIVE LEARNING USER QUERY ACCURACY (#, %)

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

there is also some human effort involved to select the "three best".

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

IV. CONCLUSIONS AND FUTURE WORK

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

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

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

REFERENCES

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

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

[3] M. S. Beach, W. Ely, and G. B. Vanderpoel. *The Ely Ancestry*. The Calumet Press, New York, New York, USA, 1902.

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

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

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

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

[8] N. Dalvi, R. Kumar, and M. Soliman. Automatic wrappers for large scale web extraction. *Proceedings of the VLDB Endowment*, 4:219–230, 2010.

[9] H. Elmeleegy, J. Madhavan, and A. Halevy. Harvesting relational tables from lists on the web. *Proceedings of the VLDB Endowment*, 2:1078–1089, 2009.

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

[11] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

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

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

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

[15] A. K. McCallum. MALLET: a machine learning for language toolkit. http://mallet.cs.umass.edu/, 2002.

[16] B. Settles. Active learning. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 6(1):1–114, June 2012.