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

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

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

Categories and Subject Descriptors: I.2.7 [**Artificial Intelligence**]: Natural Language Processing—*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 learning, active learning, grammar induction, OCRed text document, list, ontology population, HMM, Hidden Markov Model

ACM Reference Format:
Thomas L. Packer and David W. Embley, 2014. Unsupervised Training of HMM Structure and Parameters for OCRed List Recognition and Ontology Population. *ACM Trans. Knowl. Discov. Data.* 0, 0, Article 00 (0000), 43 pages.
DOI: http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

Information extraction and ontology population are areas of research concerned with building models or processes to discover information in implicitly-structured sources like text and to make the structure of that information explicit, machine-readable, and more readily usable by computing machines. Wrapper induction [Kushmerick 1997] and other machine-learning-based approaches are commonly employed to efficiently produce an extraction model or wrapper. Supervised machine-learning-based approaches are common (e.g. [Heidorn and Wei 2008], [Li et al. 2011]) and can perform well in terms of accuracy, but often require a large amount of experimentation and

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

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

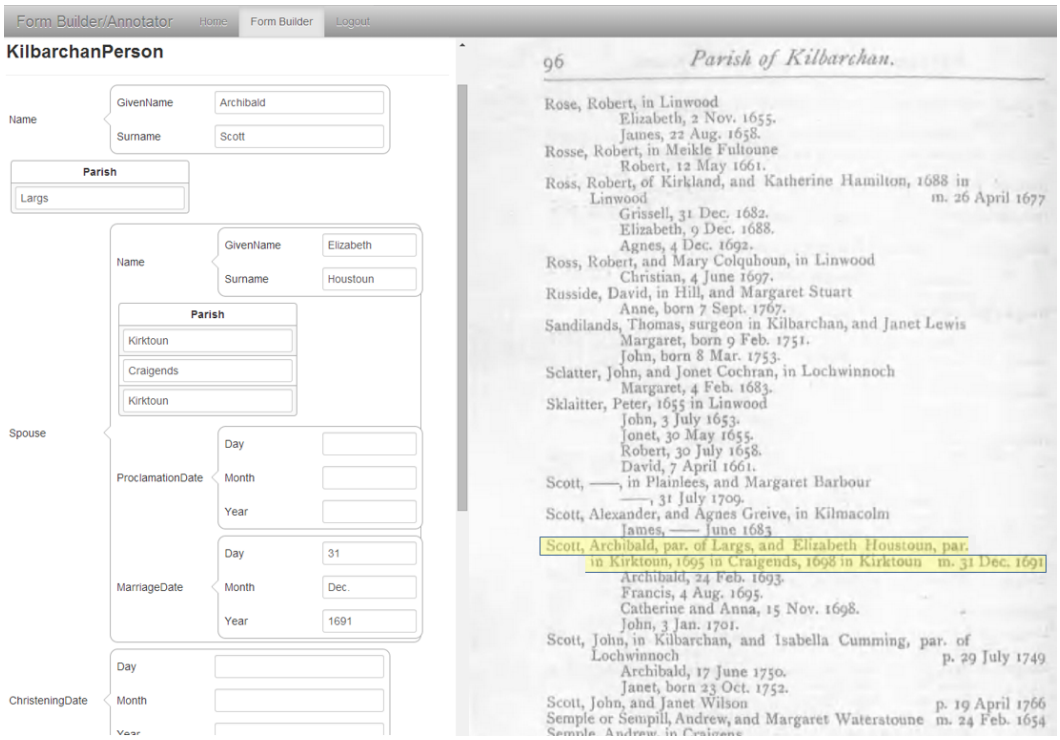


Fig. 1. *Kilbarchan Parish Register* Page and KilbarchanPerson Filled-in Form

knowledge engineering to produce an effective set of feature extractors and a large number of manually selected and annotated examples to learn well.

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

To start ListReader processing, a user selects a text document containing one or more lists of records, e.g., an OCRed collection of page images from a scanned book selected for an information application. For example, the user could select the *Kilbarchan Parish Register* [Grant 1912] for a family history application. Part of one page of this book appears in the right side of Figure 1. The user constructs a data entry form for the desired information in the left side of the user interface, e.g., the form

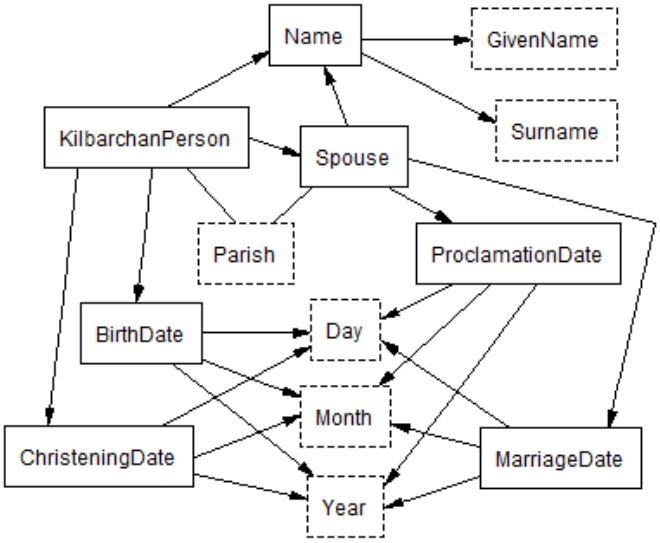


Fig. 2. KilbarchanPerson Ontology

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

Fig. 3. Labeled Record of the Highlighted Text in Figure 1

in Figure 1 before being filled in.[1] ListReader translates the form into an ontology schema, e.g., the target ontology in Figure 2. Without anything more than the given text document (e.g., the entire collection of page images of the Kilbarchan book in our example), ListReader applies an unsupervised process to automatically discover and align records, induces a simple phrase structure grammar, and trains the structure and parameters of an HMM. After ListReader sets the HMM's structure and parameters, it actively requests labels for selected strings of text from the user. ListReader highlights the strings it selects for the user to label, as Figure 1 shows. The user provides labels by filling in the data entry form.[2] Figure 1 shows the filled-in form for the highlighted text. ListReader uses the structure of the form to generate specialized labels for the field strings in the text document that specify the mapping of the strings to ontology predicates. Figure 3 shows the labels for the highlighted record. After labeling, the structure and parameters of the HMM are unchanged but some of the

[1]The construction of a form is the full extent of ListReader-required manual knowledge engineering.
[2]Filling in a form with ListReader-selected text is the full extent of required hand labeling.

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

Our approach to wrapper induction is a combination of unsupervised learning and active learning. ListReader is *unsupervised* in that it induces an HMM without labeled training data and does not alter the structure or parameters of this HMM after it starts making *active* requests of the user for labels which it receives and assigns to existing HMM states. Because of how the HMM is induced, one label from the user may apply to more than one HMM state, which greatly reduces the amount of required labeling. Furthermore, ListReader follows the spirit of active learning [Settles 2012] in that it uses this structural model to request labels for those corresponding parts of the known and unlabeled text that will have the greatest impact on the final wrapper's mapping from text to ontology, meaning the greatest increase in recall for the lowest number of hand-labeled fields.

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

We give the details of these contributions as follows. In Sections 2, 3, and 4 we describe the HMM wrapper induction process, illustrating the steps with a running example of the execution of ListReader on the 140-page *Kilbarchan Parish Record* [Grant 1912]. We explain in Section 2 how ListReader discovers record-like patterns in text in linear time and space and in Section 3 how ListReader derives the structure and parameters of an HMM from the discovered patterns—both without human supervision. In Section 4 we tell how ListReader creates the mapping from HMM states to an ontology using active sampling. In Section 5 we provide an evaluation of the performance of ListReader in terms of the precision, recall, and F-measure of the automatically extracted information, each as a function of manual field labeling cost, and compare the learning rates to a state-of-the-art statistical sequence labeler (CRF) and to a previous version of ListReader that induces regular-expression-based wrappers. In Section 6 we discuss performance issues and opportunities for future work, and in Section 7 we compare our proposed solution to related work on information extraction from lists and on unsupervised wrapper induction. Finally, we make concluding remarks in Section 8.

2. UNSUPERVISED PATTERN DISCOVERY

In its unsupervised process of pattern discovery, ListReader finds record-like patterns in the input text, produces a representation of the hierarchical field structure of these strings, and associates the major components (delimited field groups) across different types of records. In the next major step, detailed in Section 3, ListReader flattens this hierarchical structure into a state machine and sets the parameters of the HMM using statistics in the collection of parsed record patterns. As we explain in Subsection 2.1, ListReader begins to discover patterns by conflating the input text—substituting abstract word and phrase structure for strings of characters. ListReader

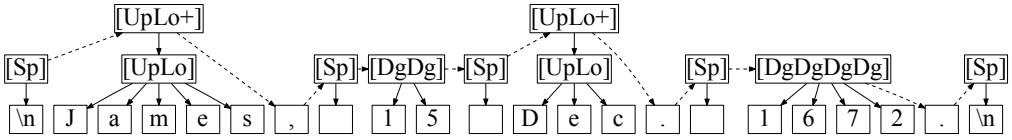


Fig. 4. Initial Parse of "`\nJames, 15 Dec. 1672.\n`"

then efficiently identifies record-like patterns in the abstract text (Subsection 2.2). Subsequently, ListReader further parses and aligns field groups within and between record patterns (Subsection 2.3). Finally, ListReader establishes the set of record and field group templates from which it constructs the HMM (Subsection 2.4).

2.1. Text Conflation

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

(1) *Split Word*: Merges two alphabetic word tokens that are separated by a hyphen and a newline into a single word symbol.
(2) *Horizontal Punctuation*: Conflates thin, horizontally-oriented punctuation characters: underscore, hyphen, en dash, em dash, and other Unicode variations.
(3) *Numeral*: Replaces each digit in a numeral with a digit designator ("`Dg`").
(4) *Word*: Replaces contiguous letters with a generic word designator that only preserves the relative order of upper-case ("`Up`") and lower-case ("`Lo`") characters. ListReader optionally preserves the full spelling of lower-case words.
(5) *Space*: Conflates normal space characters (" ") with newlines ("`\n`") using a common symbol ("`[Sp]`").
(6) *Incorrect Space*: Removes spaces that occur on the "wrong side" of certain punctuation characters because of an OCR or typesetting error, such as immediately before a period.
(7) *Capitalized Word Repetition*: Replaces each contiguous sequence of space-delimited capitalized words of any length with a single, generic symbol ("`[UpLo+]`").
(8) *Numeral Repetition*: Replaces each contiguous sequence of comma- or hyphen-separated numerals of any length greater than one with a generic symbol (e.g. "`[Dg+-]`") that only preserves the identity of the punctuation delimiter. The same delimiter must be found between every pair of numerals in a sequence. Extra spaces around the punctuation delimiter are also allowed.

The cumulative application of the conflation rules produces a sequence of small parse trees. Figure 4 shows the parse trees for the text "`\nJames, 15 Dec. 1672.\n`". The dashed line joins the root phrase symbols giving a new sequence of symbols in which ListReader looks for patterns. Not all conflation rules need be used for every book. Generally, all of them should be used that do not erase distinctions between records that should not be aligned, such as is the case when conflating lower-case words in records where different field group delimiters like "born on" and "died on" are aligned. Preventing the conflation of lower-case words is appropriate for books such as the *Kilbarchan Parish Register* [Grant 1912] which contains very little prose and whose list records are consistently structured.

2.2. Record Pattern Search

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

ListReader relies on the suffix tree to efficiently find patterns in text. A *suffix tree* is a compact data structure representing all suffixes of a text string by paths from its root to its leaf nodes. (In our running example, the text is the single long string of conflated symbols from the first conflation symbol for the first token on the first page of the book to the last conflation symbol for the last token on the last page.) Each edge in the suffix tree is labeled by the substring of symbols it represents. In addition, the information maintained for each edge includes the offsets for each occurrence of the edge's substring and a count of these occurrences. ListReader constructs a suffix tree in linear time and space using Ukkonen's algorithm [Ukkonen 1995]. It also finds and collects record-like patterns within the suffix tree in linear time and space by iterating over the edges of the suffix tree and checking for strings terminating in each edge that adhere to the properties specified for a record.

Figure 5 shows several patterns ListReader finds from the conflated text of the *Kilbarchan Parish Register*. With each pattern we also give in Figure 5 the number of unconflated strings that belong to the pattern and show a few of them—all of them for the last pattern. Observe that all of the strings of original text satisfy the properties required of records—each begins and ends with "\n" (a specified record delimiter) and includes a number or a capitalized word or both. The patterns are thus potential record patterns, and each pattern along with its candidate records forms a *record cluster*. Its pattern is called a *record template*.

2.3. Field Group Discovery

ListReader next discovers parts of records (field groups) that recur among different record clusters. These correspondences will be represented later in the HMM and used to reduce the number of necessary hand-labeled fields. The intuition is that since a field like a birth year or marriage year follows a specific field group delimiter like "born" or "m.", it can be identified and labeled the same even when found in different record clusters. The dates following "born" in the second and third record cluster in Figure 5 are all birth dates, and the dates following "m." in the last two record clusters are all marriage dates.

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

(1) *sequences of lower-case words separated by whitespace or punctuation that occur in a fixed position within a few different record clusters along with their adjacent whitespace and punctuation characters.* Requiring "few" to be more than four record clusters typically eliminates valid delimiters from consideration, and requiring less than two record clusters provides insufficient evidence for delimiter patterns. From the clusters in Figure 5, ListReader can identify the following field group delimiters of this first type: ", born ", ", and ", and " m. ". With additional supporting evidence from other clusters in the book, ListReader would also identify ", par. of " in the last record cluster in Figure 5 as a field group delimiter.
(2) *record delimiters along with whitespace and punctuation that follow or precede them.* In Figure 5, all the initial record delimiters (i.e. "\n") and both types of final record delimiters (i.e. ".\n" and "\n") are field group delimiters of this second type.

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

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

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

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

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

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

Fig. 5. A Selection of Record Clusters from the *Kilbarchan Parish Register*

ListReader constructs *field group templates* from the text appearing between field group delimiters and associates a field group template with the delimiter on its left. Field group templates consist of a field group delimiter, whose text *is not* conflated in the template, followed by one or more variations of the field group itself, whose text *is* conflated. For example, the field group template for the delimiter “, born ” in the third record cluster in Figure 5 is “, born [DgDg][Sp][UpLo+].[Sp][DgDgDgDg]” and one of its variations in the fourth record cluster in Figure 5 is “, born [DgDg][Sp][UpLo+][Sp][DgDgDgDg]”. In the third record cluster the months are abbreviations followed by a period, and in the fourth the months are spelled out and have no period. Additional variations in the *Kilbarchan Parish Record* include dates with single-digit days, dates with missing days (only months and years), and dates with day ranges, presumably when the birth day is only approximately known. The “ m. ” delimiter from the last two record clusters in Figure 5 has only one variation in the figure, but in the full *Kilbarchan Parish Record*, the dates have several variations. The initial delimiter, “\n” has the following variations in Figure 5:

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

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

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

2.4. Final Record and Field Group Template Selection

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

Figure 6 shows the new record clusters that include the record clusters in Figure 5 after grouping clusters with the minor variations. The first group in Figure 6 includes the records in the first two clusters in Figure 5, and many more—3341 altogether, which includes not only the 1296 in the first cluster and 710 in the second, but also all others that have the pattern "`\n<name>, <date>.\n`". The second cluster in Figure 6 groups the third and fourth clusters in Figure 5 as well as several other clusters that all have the pattern "`\n<name>, born <date>.\n`". The third cluster in Figure 6 groups the fifth cluster in Figure 5 with others like it, and the last cluster in Figure 6 shows the complete grouping of records for the last cluster in Figure 5. Note how the added records vary from the six in Figure 5: double-digit days, months that are not abbreviated, names with punctuation ("`M'Gregor`"), and names with a missing surname (the last two). Being complete and small enough will allow this last grouping to serve as an example for the field-group-template-splitting adjustment ListReader makes.

First, however, we explain how ListReader groups record clusters and ensures that each pattern consists of a clean sequence of field group templates. When used to parse text, we say that a field group template produces a *field group segment* as a new type of parse tree node. These nodes can be seen in the parse trees in Figure 7 for the first string of characters constituting the first record in the third cluster group in Figure 6. Except for the special "End-Segment" node, each "Segment" node includes a "Delim" node followed by a "FieldGroup" node as Figure 7 shows.

As a next step, ListReader again produces a suffix tree, working with a new input sequence composed of both field group segments wherever a field group template matches text and the original conflated text elsewhere. In Figure 7 the text matches field group templates, and thus the sequence of symbols from which ListReader constructs this second suffix tree is a sequence of "`[...-Segment]`"

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

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

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

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

Fig. 6. Record Clusters Grouped by Clean Field Group Template Sequences

symbols—"[\n-Segment][and-Segment][m-Segment][\n-End-Segment]" for the text in Figure 7. On the other hand, when the text of a potential record does not fully parse into "[...-Segment]" nodes as is the case for the three record clusters in Figure 8, the sequence of symbols is the original conflated text (e.g. the sequence of conflation symbols at the top of each record cluster in Figure 8). The first record cluster in Figure 8 fails to parse into a sequence of "[...-Segment]" symbols because of the inserted birth times, the second because of the deleted day in the date, and the third because of the OCR errors, substituting letter characters for digits.

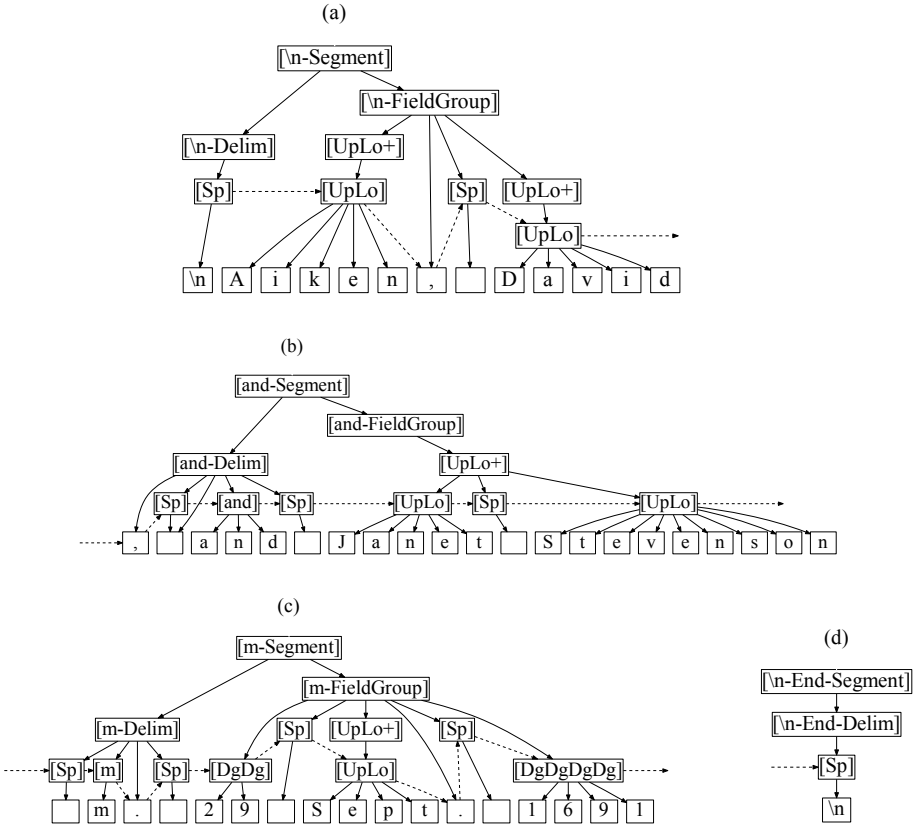


Fig. 7. Parse Trees of "`\nAiken, David, and Janet Stevenson m. 29 Sept. 1691\n`" for each Segment: (a) "`[\n-Segment]`", (b) "`[and-Segment]`", (c) "`[m-Segment]`", and (d) "`[\n-End-Segment]`"

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

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

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

Fig. 8. Record Clusters without Field Group Segment Matches

ListReader searches for record patterns in this second suffix tree as it does in the first suffix tree with one additional constraint: each record template must be composed entirely of field group segments. Thus, ListReader clusters all the text strings of

record candidates that have the same sequence of "`[...-Segment]`" templates despite any variations in the underlying text. The record clusters for the text string in Figure 7 and hundreds of other similar text strings in the *Kilbarchan Parish Record* are all grouped together in the same cluster. This clustering produces a higher level of text abstraction and a better identification of basic patterns from which to build HMM recognizers (e.g. the higher-level record patterns in Figure 6 rather than the lower-level record patterns in Figure 5). Furthermore, rejecting some clusters like those in Figure 8 helps ListReader focus on major patterns so that it does not need to create and execute HMM recognizers for the long tail of patterns that occur infrequently or happen to be exceptions to the general typeset text patterns of a book. Rejecting patterns does not mean, however, that the information in these rejected patterns will not be extracted. When we explain how ListReader constructs HMM recognizers in Section 3, we will also show that the HMMs constructed allow for deviations from the basic patterns for insertions, deletions, and substitutions, respectively like inserted birth times, missing days in dates, and OCR-error character substitutions in Figure 8. In essence, ListReader discovers the basic "clean" patterns in a book including a limited number of variations as a phrase structure grammar and also accommodates exceptions to basic patterns and larger variations with the HMM derived from that phrase structure grammar.

In grouping variations into basic clean patterns, ListReader's procedure can cluster field group templates that vary widely and therefore that a single linear HMM cannot reasonably accommodate. In Figure 6, for example, the four field group templates under the first "`[\n-Segment]`" are all quite different—varying in length (10, 5, 2, and 16 symbols, respectively) and content (respectively: given name and date, surname followed by given name, place, and two consecutive full names in two variations). To model widely differing patterns in a single high-level cluster, ListReader partitions patterns into similarity groups and for each group, selects a representative pattern as the basis for generating an HMM recognizer. We choose this approach of accounting for field group template variations instead of either creating a separate HMM recognizer for each variation or merging all the variations into a single union of parallel HMM recognizer. Using all possible variations is less desirable for two reasons: it significantly increases the running time of execution, which is quadratic in the number of states, and it significantly increases the variations that have to be hand-labeled. Merging all variations into a single pattern is probably hard to do without more domain knowledge than we expect the user to provide. Therefore, we choose to select a representative sample of field group variations from which ListReader builds its HMM recognizers.

In selecting field group template variations to be representatives, ListReader must be careful to observe significant differences and ignore minor variations. It therefore estimates the "distance" between variations and separates those that vary more than a pre-specified threshold and groups the rest with the separated templates such that they partition the space of field group template variations. Then for each block of the partition, ListReader selects a "best" field template representative. From each representative ListReader generates a component of the HMM, as we explain in Section 3.

To measure the distance between pattern variations, ListReader uses a normalized Levenshtein edit distance: the minimum number of word insertions, deletions, and substitutions to transform one pattern to another divided by the average of the lengths of the patterns. Figure 9 shows the three pattern variations and their instances for the initial field group segment of the last record cluster in Figure 6. The normalized edit distance between the first two pattern variations, "`\n[UpLo] , [Sp] [UpLo]`" and "`\n[Up] ' [UpLo] , [Sp] [UpLo]`", is $2/((5+7)/2) = 0.333$—two insertions, "`[Up]`" and "`'`" into the first pattern (length 5), yields the second pattern (length 7). The normalized

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

```
[\n[Up]'[UpLo],[Sp][UpLo]]
(count 1, length 7)
\nM'Gregor, John
```

```
[\n,[Sp][UpLo]]
(count 2, length 4)
\n, William
\n, Robert
```

Fig. 9. Initial Field Group Template Instances in the Last Record Cluster in Figure 6 Grouped by Pattern Variation

edit distance between the first and the third is 0.222, and between the second and third is 0.545. If the threshold is 0.5, the second and third pattern variations should be in separate groups. The first can be grouped with either the second or the third, and should be with the third in this example since it is "closer."

When a group has more than one pattern variation, ListReader selects one to be the *representative* for the group. ListReader constructs its HMM from a combination of all the representatives, one for each group. To select the best representative, ListReader computes a representative score for a pattern variation by multiplying the pattern's length in symbols by its instance count and chooses the pattern variation with the highest score as the representative for the group. For the pattern variations in Figure 9 the scores are 20×5, 1×7, and 2×4 for the first, second, and third group respectively. Thus, for our example in Figure 9, "`\n[UpLo]` , `[Sp][UpLo]`" is the representative for the group consisting of the first and third pattern and "`\n[Up]'[UpLo]` , `[Sp][UpLo]`" is the representative of its own group. The representative score is motivated by observing that the score for a pattern is the number of tokens the pattern matches in the full text—more is likely to be better. In preliminary experiments on the *Shaver-Dougherty Genealogy*, ListReader's representative-selection procedure improved precision, recall, F-measure, and reduced the number of required hand-labelings compared to two other policies: one that selects the single longest pattern variation among all variations and one that selects the longest pattern for each group of pattern variations. Also, changing the edit distance cut-off values between 0.1 and 0.9 did not significantly affect the final evaluation scores, nor did normalizing by the length of only one of the compared patterns.

Figure 10 shows the representative templates along with an instances to illustrate them for the remaining field group template sequences in Figure 6. These template sequences also have some field-template variations, but perhaps different from what might be expected. The third cluster of records about couples and marriage dates does not have enough pattern variation to warrant breaking any of the field templates into groups. Date variations following the "`m.`" such as double-digit vs. single-digit days and abbreviated vs. non-abbreviated months are at most an edit distance or two apart (not

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

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

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

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

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

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

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

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

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

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

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

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

Fig. 10. Template Representatives of First, Second, and Third Record Cluster in Figure 6 (The "?" between "Mary" and "Archibald" is a character error; on the original page in the Kilbarchan book it is an m-dash.)

enough when compared to the number of symbols in the "`[m-Segment]`" template). In addition, in these marriage records name patterns and patterns of record termination with no period are highly consistent across more than 100 pages in the *Kilbarchan Perish Record*. On the other hand, the first record cluster, with the template sequence "`[\n-Segment][\n-End-Segment]`", does break into several groups as Figure 10 shows. Like the third record cluster, the second cluster's "`[\n-End-Segment]`" template breaks into two groups (normalized edit distance: 1/1.5 = 0.667), but neither of its two preceding field group templates has enough variation to cause a break into groups.

3. HMM CONSTRUCTION

An HMM is a probabilistic finite state machine consisting of a set of hidden states S, a set of possible observations W, an emission model $P(w|s)$ associating a state with a set of observable events, and a transition model $P(s_t|s_{t-1})$ associating one state with the next. States are initially "hidden" and must be inferred during application of the HMM from the observable events in the text. In our work, each event is a word-sized chunk of text (a token), including alphabetic words, numerals, spaces including newlines, and

punctuation characters. Inferring the correct state associated with each word token is the main task done in extracting information from the text and is guided by the parameters of the HMM. Using the Viterbi algorithm, ListReader selects the most probable sequence of states given the words of the input text and the HMM's parameters. The emission model is a categorical distribution—a table of conditional probabilities indicating which observation w can be emitted[3] from which hidden state s and with what probability given s. The transition model is also a categorical distribution—a table of conditional probabilities indicating which hidden state s_t at position t can follow which other hidden state s_{t-1} at position $t-1$ and with what probability given s_{t-1}. The two kinds of probabilities are the parameters of the HMM. The set of states and the transitions that have non-zero probabilities in the transition model determine the structure of the state machine of the HMM. The processing described in Section 2 provides what ListReader needs to produce the set of hidden states and both the transition and the emission model for our application.

The HMM that ListReader constructs has two levels of structure, page-level and record-level, that are connected by transitions. The record-level states belong to record templates that are connected to each other and to the page-level states of the HMM. In Subsection 3.1, we explain how ListReader generates states for field group templates and how it labels states, providing them with syntactic and semantic IDs. (In Section 4 we tell how ListReader transforms these IDs into labels that map text associated with the HMM's hidden states to an application ontology.) In Subsection 3.2, we discuss transition and emission models for field group templates and say how ListReader sets parameters at the level of field group templates. Finally, in Subsection 3.3 we explain how ListReader finishes the transition model connecting HMM fragments for field group templates to each other to form HMM components for record templates and connecting record-template components to page-level HMM states and setting page-level transition and emission parameters.

3.1. Field Group Template State Generation

ListReader transforms each field group template representative into a linear sequence of HMM states, one HMM state for each word token in the parse tree of the field group segment. (In Figure 7 the dashed arrows show the sequence of word tokens considered for HMM construction.) Consider the representative templates in Figure 10 and particularly, for example, the representative template

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

for the "`[m-Segment]`" (second from the bottom in Figure 10). The generated HMM fragment for this representative template has ten states, one for each word token in the representative template. Figure 11 shows these ten states. In addition to one state per word token in a field group template, ListReader generates an *insertion* state between every pair of consecutive word states. Figure 11 shows these insertion states as empty nodes. These insertion states allow for inconsistent punctuation and noise in pattern delimiters and for sparse comments such as when the time of birth is given as the first cluster in Figure 8 shows. Figure 11 also shows the transitions among the states for the representative template (albeit, as yet without the transition probabilities). The main transitions form a straight line through the template; the transitions to and from insertion states allow for text-addition deviations from a typical record; and the transitions that skip over word states allow for text-omission anomalies. The

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

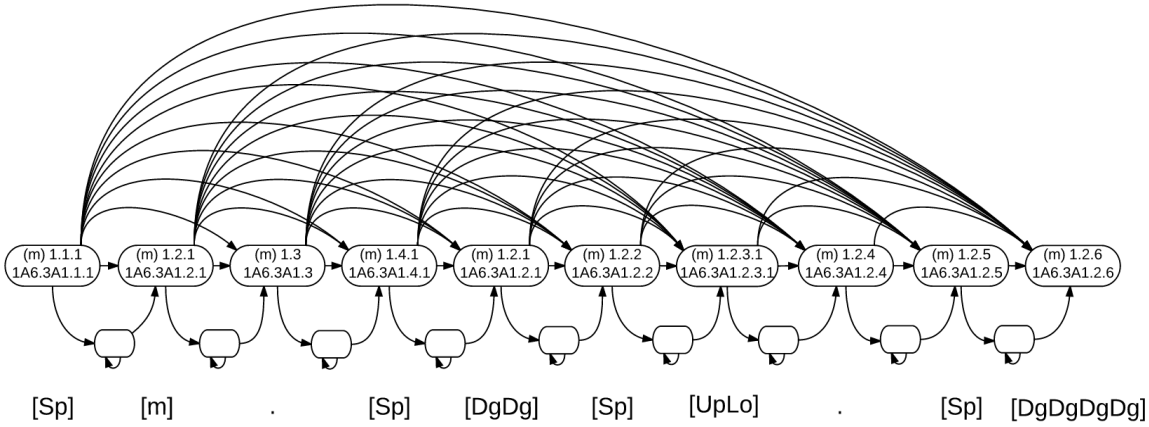


Fig. 11. HMM States and Connecting Transitions for "`[m-Segment]`" whose Representative Template is "`[Sp][m].[Sp][DgDg][Sp][UpLo].[Sp][DgDgDgDg]`"

missing days in the dates in the second cluster in Figure 8 is an example of when the HMM should skip states by taking a deletion transition. Patterns with inconsistencies and exceptions are those that tend to be discarded when ListReader culls its set of discovered patterns keeping only record patterns that are a clean sequence of representative field-group-template segments. Insertion states and deletion transitions help overcome deficiencies caused by inconsistencies and exceptions.

Figure 11 further shows that ListReader assigns each state two IDs—a *semantic ID* and a *syntactic ID*. Each state's syntactic ID (the second identifier in the nodes of Figure 11) is fundamentally a dot-delimited sequence of numbers representing the path in the parse tree from root (a record node that groups all segment nodes of a record template) to word token (the word token for which the state is being created). Each number in this path indicates the order of the node among its siblings in the parse tree. Consider, for example, the parse trees in Figure 7 with "`[Record]`" added as the root node forming one parse tree for the record template. Then the path to "`Aiken`" is

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

and the path to "`1691`" is

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

To make the syntactic ID of each HMM state functionally complete within the HMM, ListReader assigns "alternative" numbers ("A" numbers) to both the `[Record]`-level number and the `[...-Segment]`-level number in a path. The `[Record]`-level number identifies the alternative among all record templates discovered by ListReader, each of which implies a certain set of field group templates in a certain order. This record-level alternation number, along with the parse tree numbers, makes the syntactic ID of each node unique within the complete HMM. In our run of ListReader on the *Kilbarchan Parish Record*, the "`[[\n-Segment][and-Segment][m-Segment][\n-End-Segment]]`" record happens to have been the 6^{th} alternative record template found. The `[...-Segment]`-level number identifies which field-group alternative pertains to the

state—1 in our example since the "[m-Segment]" in Figure 10 has only one alternative. Thus, in Figure 11 the tenth state, for example, has the syntactic ID "1A6.3A1.2.6", with [Record]-level alternative "A6" and [...-Segment]-level alternative "A1".

The semantic ID of an HMM state identifies the field group template to which the states of the field group apply and also the position of the token within the field group template. A semantic ID is the suffix of the syntactic ID starting with the field group segment's alternation number. (We prepend the field group template's identifying name parenthetically to the semantic ID for human readability. In Figure 11, since all the states belong to the "[m-Segment]" field template, they all begin with "(m)".) The semantic ID, therefore, represents both a type of field group segment and a token's position within that field group segment.

The semantic ID of a state is purposefully *not* unique within an HMM. States that share a semantic ID (and in turn the words they match) should be labeled the same because they have the same relationship with the primary object of their respective records. For example, all "m. <date>" constructs in the entire *Kilbarchan Perish Record* are marriage dates and should be labeled as such regardless of which record template in which they are found. (In our run of the Kilbarchan book, seven different record templates include the "[m-Segment]" field group template, two of which are in Figure 6.) ListReader should request only one label from the user for all states that share the same semantic ID—and therefore request only one label for the many hundreds of "m. <date>" strings in the Kilbarchan book. ListReader carefully infers semantic IDs and therefore carefully assigns field group alternation numbers. ListReader assigns the same field group alternation number to field group templates that (1) are of the same type (e.g. "and" and "m" are different) and (2) contain the same conflated field group words (e.g. "[UpLo]" and "[UpLo] [Sp] [UpLo]" are different). Therefore, HMM states with the same field group alternation number will have the same semantic ID if they are in the same position within their respective field group segments, even when those field groups appear in different record templates. This semantically ties the HMM states together that refer to the same field group templates and, in active sampling (Section 4.1), prevents the user from labeling more than one example of that field group. For example, the field group template "[m-Segment]" matching " m. 29 Sept. 1691" in the third cluster of Figure 6 also matches " m. 18 Dec. 1672" in the fourth cluster and will be assigned the same final labels because they will first be assigned the same semantic IDs, despite being in different positions in two different record templates.

3.2. Field Group Template Parameter Setting

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

A substitution is a token in one record that does not exactly match the corresponding token in another record. For example, if an HMM fragment were built for the text "\nJames, 15 Dec. 1672.\n", we still expect the fragment to match text like "\nJames, 15 Dee, 1672.\n", despite the comma replacing the period and the "e" replacing the "c" (likely due to noise in the document causing OCR errors). ListReader allows for substitutions using both conflation of text and smoothing in the emission model: "[UpLo]" conflates "Dee" as well as "Dec", and the emission model settings allow for alternatives in fixed text—punctuation and delimiter text. The emission model

of each record-level state is set with the word-level conflated text for that state with a count of 1.0, unless the state is part of a delimiter in which case ListReader uses the non-conflated text, e.g. "`[m]`" instead of "`[Lo]`". Emission parameters for conflated tokens belonging to numeral and alphabetic word character classes are smoothed with small, fractional pseudo-counts to allow for any other numeral or alphabetic words with low probability (lower for words outside of the character class of the original text). For example, in Figure 12 the word with conflated text "`[DgDgDgDg]`" receives a count of 1.0 for "`[DgDgDgDg]`", a pseudo-count of 0.01 for "`[Dg]`", "`[DgDg]`", "`[DgDgDg]`", and "`[DgDgDgDgDg]`" and a pseudo-count of 0.001 for "`[UpLo]`", "`[LoUp]`", "`[Up]`" and "`[Lo]`". In general, ListReader's construction of emission models promotes better alignment of similar words, especially words of the same character class, despite the small amount of training data provided and despite possible OCR errors and other variations. Similarly, as Figure 12 shows, ListReader adds pseudo-counts of spaces for the two kinds of internal space it encounters, (" " and "`\n`"), thus accommodating line breaks in the middle of a record where spaces usually appear. Finally, all record-level state emission models except for "`[Sp]`" and record delimiters ("`\n`" for our running example) receive a pseudo-count of 0.0001 for every other word in the document. We have omitted these smallest parameters from the figure for simplicity. After collecting the counts for a state's emission model, ListReader sums the counts and divides the various counts by the sum to establish normalized emission probabilities that sum to one for each state's emission model as Figure 12 shows.

A deletion is a sequence of one or more tokens of a record template that are missing in the text that should otherwise match that record template. For example, although the HMM fragment in Figure 11 is for text like "`\nJames, 15 Dec. 1672.\n`" we expect the HMM to be flexible enough to match text like "`\nJames 15 Dec. 1672\n`" (with some of the punctuation missing) or like "`\nJames, Dec. 1672.\n`" (with the day in the date missing). To accommodate deletions, states in the HMM that are not adjacent in training data should become adjacent during execution. To allow for deletions during unsupervised training, the transition model of each pair of adjacent states receives a full count of 1.0, while the transition model of each pair of non-adjacent states receives a pseudo-count of $1/30$ if a pair of states satisfies our deletion constraint and zero otherwise. The deletion constraint requires an ordering on states: the second state must follow the first state within a training record, regardless of how far apart the words are. For example, in Figure 11 the "m" precedes the double-digit number for the day in the date, so the transition from the state representing "m" to the state representing the double-digit day receives the $1/30$ pseudo-count. But the reverse transition (from the double-digit state to the "m" state) would receive a zero pseudo-count. Proper order is determined algorithmically by comparing the parse tree numbers in the syntactic IDs of the two HMM states in question. The algorithm checks to see that the two states have ancestor numbers that are correctly ordered: e.g. states "1A6.3A1.1.3" for "m" and "1A6.3A1.2.1" for the double-digit state are correctly ordered because they have the same record alternation number ("A6") and the same field-group-segment alternation number ("A1") in the same position (3), and their field group parse tree numbers are in the correct order ($3.1.3 < 3.2.1$). But the reverse order would not be allowed. Field-group-segment alternation number can be different as long as their positions are also different. Figure 12 shows the probabilities on the deletion transitions (which also depend on out transitions to insertion states, discussed next).

An insertion is a sequence of one or more tokens appearing in text that should match a record template but which did not appear within the training text of that record template. For example, although the HMM fragment in Figure 11 is for text like "`\nJames, 15 Dec. 1672.\n`" we expect the HMM to be flexible enough to match text like "`\nJames, 15. Dec. 1672..\n`" (containing two extra pe-

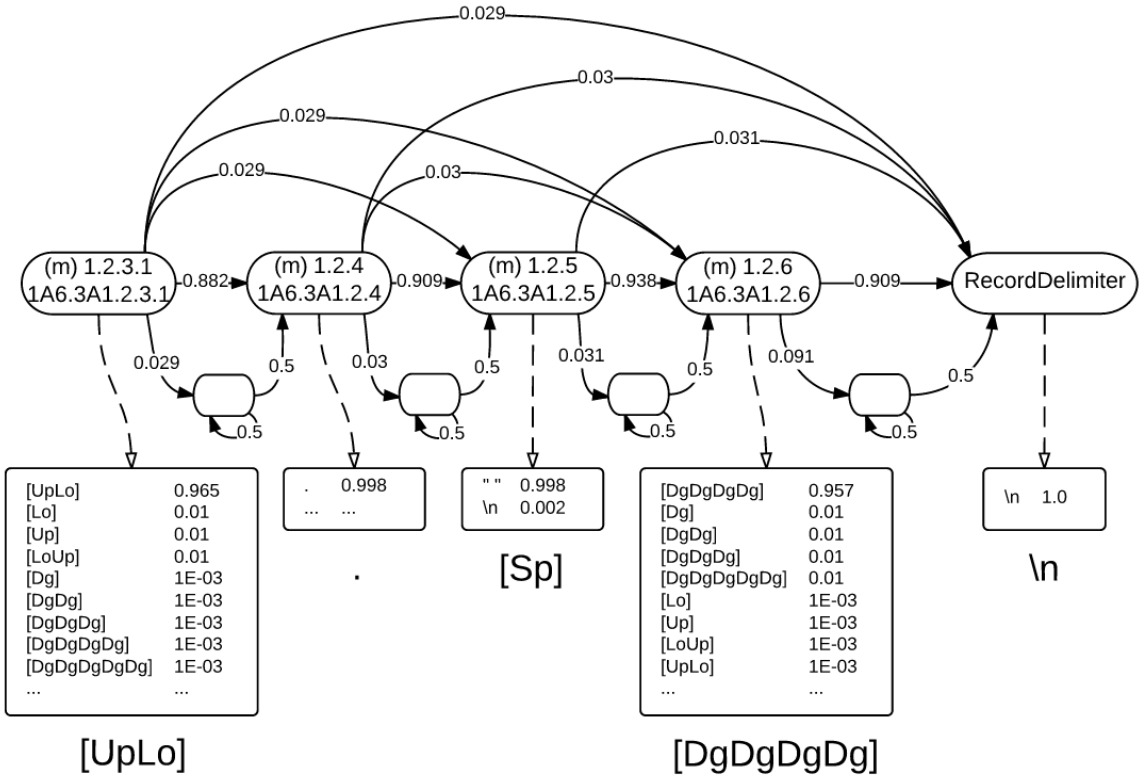


Fig. 12. HMM Transition and Emission Models for the Last Four States of "`[m-Segment]`" and the Single State of "`[\n-End-Segment]`" for the Last Record Template in Figure 10

riods, perhaps because of noise in the text). It should also match text of occasional comments that may not have been seen in the training text, so that the HMM fragment for "`\nJames, 15 Dec. 1672.\n`" would also match the text "`\nJames, 15 Dec. 1672. (father dead)\n`" in which the author is adding a comment saying that a father died before the child's birth. To accommodate as-yet unseen additions in a pattern, ListReader generates insertion states between record-level states s_{t-1} and s_t. Although not shown in our HMM figures, each insertion state has as its syntactic label the concatenation of the prior and subsequent states' syntactic label $(s_t + s_{t-1})$ and as its semantic label the concatenation of the two states' semantic labels. ListReader sets counts for three new transitions per insertion state: one to the insertion state from the prior state: s_{t-1} to $(s_t + s_{t-1})$, one self-transition for possible additional insertions: $(s_t + s_{t-1})$ to $(s_t + s_{t-1})$, and one from the insertion state to the subsequent state: $(s_t + s_{t-1})$ to s_t. The pseudo-count is the same for all three transitions: $n/30$, where n is 1, 2 or 3 depending on the likelihood of insertion at that location given prior knowledge of the behavior of insertions in list-like text: $n = 3$ next to record delimiters, $n = 2$ next to field group delimiters, and $n = 1$ everywhere else. In Figure 12, for example, the transition probability derived from the pseudo-count for the transition to the insertion state next to the record delimiter is greater than the probabilities on the transitions to other insertions states, which models the expectation that an insertion at the end of the record is more likely than in the middle.

3.3. Connecting the Pieces

As Figure 13 shows, ListReader generates page-level states for the beginning (*PageBeginning*) and ending (*PageEnding*) of each page and connects them to states for non-list text (*NonList*) and for list-record text (*RecordDelimiter*), the beginning state for all record-template HMMs. Figure 13 also shows how ListReader connects its record-delimiter state to every HMM record template—all 47 of them for our example run of the *Kilbarchen Parish Record*. One of the record-template HMMs is open, schematically showing the interconnections of the HMM fragments for

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

the second record template in Figure 10. Notice that the field group template "`[\n-End-Segment]`" has two representative templates, one for "`.\n`" and one for "`\n`". Whenever a field group template has multiple representative templates, ListReader generates parallel HMM fragments, one for each identified representative template. Each field group template has an HMM fragment of the form of the field-template HMM for "`[m-Segment]`" in Figure 11. Schematically, Figure 13 shows for each HMM field group template only the initial and final states along with the entry and exit insertion states. Each HMM fragment requires connections to all prior and subsequent HMM fragments as Figure 13 shows. When consecutive parallel HMM fragments occur as they would for the first record template in Figure 10, the HMM fragments are connected in a cross-product fashion. Thus ListReader would generate sixteen transitions to connect the pieces of the first record template in Figure 10, which contains four HMM fragments for "`[\n-Segment]`". Each of the four representative field group templates has four out-transitions, three going to the two end-record HMM fragments and one going directly to the record delimiter as in Figure 13.

ListReader creates transitions from the *RecordDelimiter* state to the start states of each of the initial field group templates for every record template it discovers. ListReader sets the count for each of these transitions equal to the sum of the sizes of the clusters in the group for the representative template being modeled. For the HMM of the record template for the group consisting of the first and third clusters in Figure 6, for example, the count would be 22 (20 for the first cluster plus 2 for the third). The second cluster in Figure 6 is in a group of its own and has the count 1. Counts for transitions from the *RecordDelimiter* state to an initial insertion state are $3/30$ as specified earlier for an insertion state adjacent to a record delimiter. The count for the out-transition to the *NonList* state is the number of locations in the book where a record delimiter is followed by text that was not identified as being part of a list record, and the count for the out-transition to the *PageEnding* state is the number of pages in the book that end in a record delimiter. Summing these counts and normalizing them yields the transition probabilities for transitions emanating from the *RecordDelimiter* state. Figure 13 shows eight of these transition probabilities for our example run of the *Kilbarchen Parish Record*, with two of the edges in the figure contain parameters for both a transition to an insertion state and to a non-insertion state separated by a comma.

ListReader generates states and transitions for its page-level model as Figure 13 shows. The emission model of *PageBeginning* and *PageEnding* are fixed to contain only the special character that ListReader artificially inserts into the text sequence at the beginning and ending of each page to represent page breaks. The emission model of *RecordDelimiter* is fixed to contain the set of allowable record delimiters, which currently contains only the newline character. For these fixed emission models, the probability of the allowable character is 1.0 and all other probabilities are 0.0. The emission model of *NonList* state is not fixed. Rather, it is set as the MLE estimate of all word

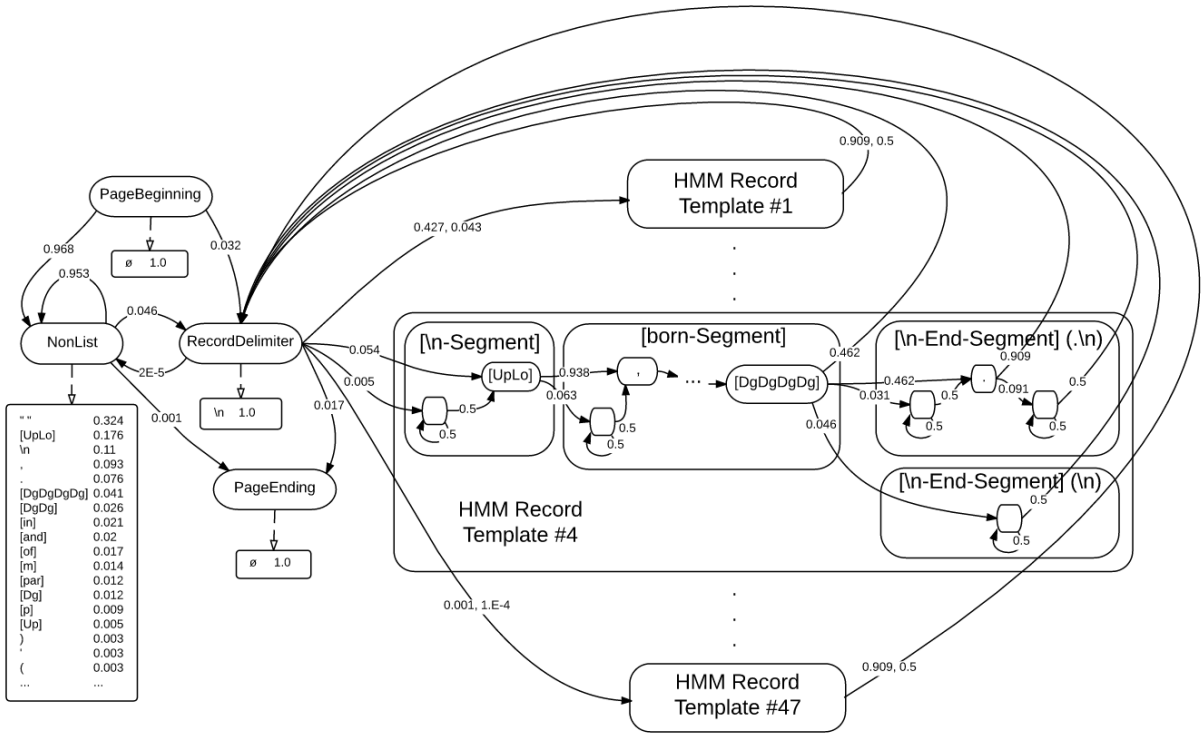


Fig. 13. Schematic Diagram of ListReader-generated HMM

tokens in the input text that were not covered by any candidate records during unsupervised grammar induction. The emission model for the *NonList* state in Figure 13 lists several of these word tokens and their probabilities based on actual occurrence counts in our run of ListReader on the *Kilbarchan Parish Record*. The most frequently occurring word token not included in candidate records is the space character with 32.4% of the uncovered tokens and "`[UpLo]`" with 17.6%. Three of many tokens that appear only once are "mile", "are", and "about"—not shown in Figure 13 along with many others. We train the emission model of the *NonList* state on unlabeled data and the emission models of list states on labeled data (specifically automatically-labeled data). These two sets of states (list and non-list) can be seen as a binary classifier, predicting a "positive" and a "negative" class. We justify our approach to training our HMM from mixed labeled and unlabeled data by citing Elkan and Noto ([Elkan and Noto 2008]) who show that for binary classifiers, "under the assumption that the labeled examples are selected randomly from the positive examples ... a classifier trained on positive and unlabeled examples predicts probabilities that differ by only a constant factor from the true conditional probabilities of being positive." We also smooth the emission model of the *NonList* state using small Dirichlet priors to allow any word to appear, even those not appearing in the training data.

The parameters for the other transitions among the four page-level states are also trained using MLE from the records discovered during grammar induction. For the emanating transitions of the *PageBeginning* state, for example, if there were 100 pages of input text and 10 of the pages began with list text and 90 with non-list text, then the

transition from *PageBeginning* to *RecordDelimiter* would receive a count of 10 while the transition from *PageBeginning* to *NonList* would receive a count of 90. The transition model is also smoothed with small Dirichlet priors to allow any reasonable transitions that were not seen in the parsed text such as a transition from *PageBeginning* to *PageEnding*, allowing for an empty page or a page consisting only of a picture. For the *NonList* state, the count for the transition to the *RecordDelimiter* state is the number of instances in the book where ListReader identified text outside a list immediately before a list; the count for the self loop returning to the *NonList* state is number of word tokens in the book that are not covered by record templates and do not immediately precede a list; and the count for the transition to the *PageEnding* state is the number of pages in the book that do not end with a list. ListReader normalizes counts for each emanating state, producing the transition probabilities.

Figure 13 shows the transition probabilities for our example run of the *Kilbarchan Perish Record*. Altogether, the full ListReader-generated HMM for our example run of the *Kilbarchan Parish Record* has 1,805 states and 3,717 transitions.

4. LABELING AND FINAL EXTRACTION

To populate an ontology with extracted information, ListReader (1) obtains labels from a user for HMM states and (2) maps labeled text to the ontology. To obtain labels, ListReader actively and selectively requests labels that associate HMM states with elements of the ontology, as explained in Subsection 4.1. ListReader then applies obtained state-label knowledge to extract information from throughout the input text and map it to the ontology, as explained in Subsection 4.2.

4.1. Active Sampling

Active sampling consists of a cycle of repeated interaction with the user. On each iteration of the loop, ListReader selects and highlights text that matches part of the HMM, and the user labels the fields in highlighted text. Labeling consists of the user copying substrings of the ListReader-selected text into the entry fields of the data entry form in ListReader's UI as Figure 1 shows. ListReader then accepts the labeled text via the web form interface and assigns labels to the corresponding HMM states, which completes that part of the HMM and enables it to become a "wrapper" that extracts information from the text and maps it to the ontology as we explain in Subsection 4.2.

The active sampling cycle is a modified form of active learning, focusing on the "active sampling" step and performing practically none of the "model update" step, just as in [Hu et al. 2009]. The HMM training ListReader does is fully unsupervised—no HMM structure or parameter learning takes place under the supervision of a user either interactively or in advance. Label renaming is the only change ListReader makes to the HMM during active sampling. In each cycle, ListReader actively selects the text for labeling that maximizes the return for the labeling effort expended. To initialize the active sampling cycle, ListReader applies the HMM to the text of each page in the book. It labels the strings that match each state with the state's semantic ID. ListReader saves the count of matching strings for each semantic ID. It also records the page and character offsets of the matching strings throughout the book and their associated semantic IDs. ListReader uses the page and character offsets when highlighting a span of text in the UI for the user to label. ListReader selects a span of text on each iteration of active sampling using a query policy (explained next) that is based on the counts of matching strings for each semantic ID.

The string ListReader selects as "best" is a string that matches the HMM fragment with the highest predicted return on investment (ROI). The ROI can be thought of as the slope of the learning curve: higher accuracy and lower cost produce higher ROI. The HMM fragments considered are HMM record templates or contiguous parts

thereof (e.g. the HMM fragment for "[m-Segment]" illustrated in Figure 11). When more than one string matches the best HMM fragment, ListReader selects the first one on whichever page contains the most matches of that HMM fragment. ListReader computes the predicted ROI as the sum of the counts of the strings matching each state in the candidate HMM fragment divided by the number of states in the HMM fragment—that is, the average match-count per state. Querying the user to maximize the immediate ROI tends to maximize the slope of the learning curve and has proven effective in other active learning situations [Haertel et al. 2008]. Once the user labels the selected text, ListReader removes the counts for all strings that match the corresponding states or that share the semantic IDs of labeled states, recomputes the ROI scores of remaining states, and issues another query to the user.

In our example run of the *Kilbarchan Parish Record*, ListReader selects the highlighted text in Figure 14. Its HMM record template is composed of the first representative for the first "[\n-Segment]" field group template and the first representative for the first "[\n-End-Segment]" field group template in Figure 10. There are nine matching states in this HMM record template, one for each word-level, non-record-delimiter symbol, "[UpLo] , [Sp] [DgDg] [Sp] [UpLo] [Sp] [DgDgDgDg] .". The hit count for the strings matching each state are:

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

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

When one HMM state receives a user-supplied label, all states sharing the same semantic ID receive the same final label. In the example in Figure 14 the user would label "Marie" as *KilbarchanPerson.Name.GivenName*, "17" as *KilbarchanPerson.ChristeningDate.Day*, "June" as *KilbarchanPerson.ChristeningDate.Month*, and "1653" as *KilbarchanPerson.ChristeningDate.Year*. And, since given-name and date fields in other christening record-templates have the same semantic IDs, these fields are also labeled—thousands of them due to the date variations (abbreviated/non-abbreviated months and single-digit/double-digit days) that appear in the *Kilbarchan Perish Record*. Furthermore, all delimiters are implicitly labeled whenever a user labels the fields in a record as the text between labeled fields and preceding the first labeled field and following the last labeled field. In the example in Figure 14, the user implicitly labels four delimiters: the comma and space between the name and the day in the date, the two spaces within the date, and the period following the year. The states for delimiters also have semantic IDs as Figure 11 shows, so ListReader propagates the labels to all other states with identical semantic IDs—those that have the same delimiter in the same position in the same field group template.

ListReader's label propogation across semantic IDs minimizes the user's labeling effort during active sampling. As an example, Figure 15 shows ListReader's second

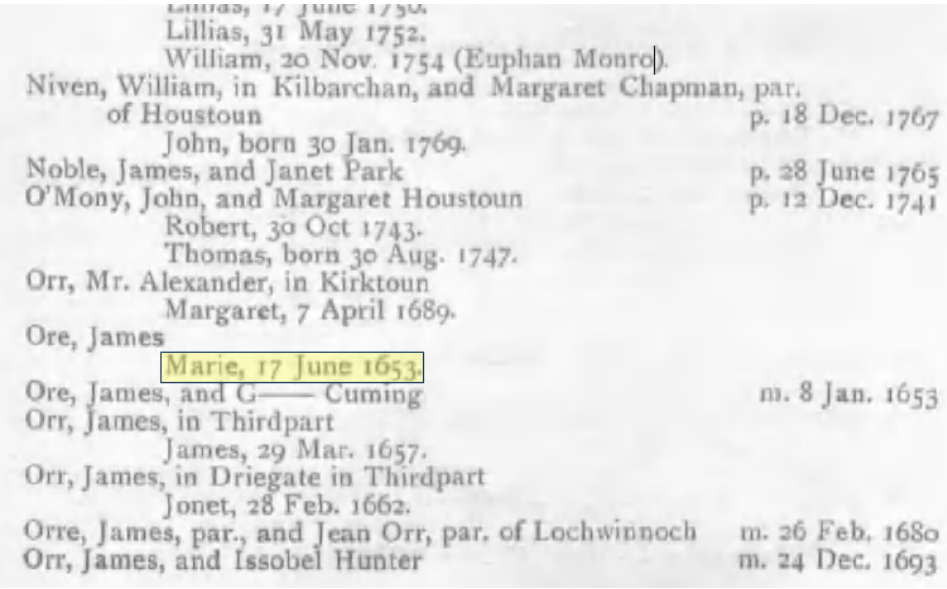


Fig. 14. First Active-Sampling User Query

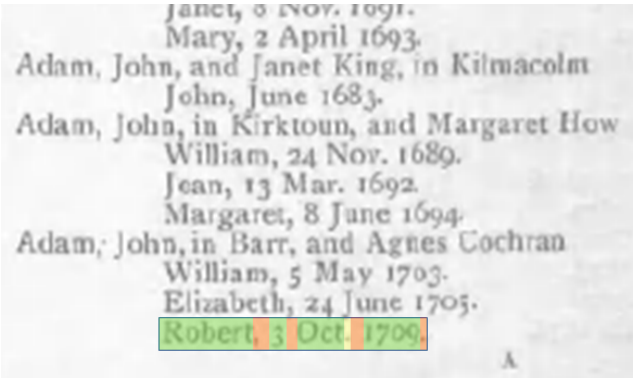


Fig. 15. First Active-Sampling User Query Requiring Only Partial Labeling

active-sampling query for our example run of the *Kilbarchan Parish Record*. The highlighting is multicolored: green for previously labeled fields (the *GivenName* "Robert", the *ChristeningDate.Day* "3", the *ChristeningMonth* "Oct", and the *Christening.Year* "1709"); red for previously labeled delimiters (", ", " ", " ", and "."); and yellow for unlabeled text (the period following "Oct" in Figure 15). ListReader does not know, by what it has so far learned, whether the period following "Oct" belongs to the *Month* field or to the delimiter between "Oct" and "1709". At this point, the user should copy "Oct." into the *KilbarchanPerson.Christening.Month* form field to label the "." state following the "`[UpLo]`" state now known as the *KilbarchanPerson.ChristeningDate.Month* state in the HMM fragment as being part of the month field.

As our Kilbarchan example shows, active sampling is impactful from the first query. Furthermore, it improves recall monotonically as it does not back-track or reverse labeling decisions from one cycle to the next. Compared with typical active learning

[Settles 2012], it is not necessary for ListReader to induce an intermediate model from labeled data before it can become effective at issuing queries. This would be true even if ListReader did update the HMM during active learning cycles, although it would necessitate ListReader having to apply the HMM again on every cycle, which currently it avoids. Furthermore, ListReader need not know all the labels at the time of the first query. Indeed, it starts active sampling without knowing any labels. The query policy is similar to processes of novelty detection [Marsland 2003] in that it effectively identifies new structures for which a label is most likely unknown. Furthermore, the wrapper can be induced for complete records regardless of how much the user annotates or wants extracted, and ListReader is not dependent on the user to identify record- or field-delimiters nor to label any field the user does not want to be extracted.

4.2. Mapping Data to Ontology

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

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

ListReader maps the empty HTML form to an ontology schema that may contain a number of conceptual distinctions including any of the following. (1) textual vs. abstract entities (e.g. *GivenName*("Archibald") vs. *KilbarchanPerson*($Person_1$) in Figure 1, where $Person_1$ is an object identifier); (2) 1-many relationships in addition to many-1 relationships so that a single object can relate to many associated entities or only one (e.g. a *KilbarchanPerson* object in Figure 2 can relate to several *Parish*es but only one *ChristeningDate*—the arrowhead in the diagram on *ChristeningDate* designating functional, only one, and the absence of an arrowhead on *Parish* designating non-functional, allowing many); (3) n-ary relationships among two or more entities instead of strictly binary relationships (e.g. if a user wants to associate dates of residence in a parish along with the parish name yielding a ternary relationship among *KilbarchanPerson*, *Parish*, and *ResidenceDates* in Figure 2 and being designated by a double-column multiple-entry field with *ResidenceDates* along side of *Parish* 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. *KilbarchanPerson.Spouse.MarriageDate.Day* in Figure 2); (5) concept categorization hierarchies,

including, in particular, role designations (e.g. if a user wants to designate roles for some Kilbarchen persons who have duties in the parish, such as a priest or an alter boy); and (6) a non-tree ontology structure (object sets can be shared among multiple relationship sets). This expressiveness provides for the rich kinds of fact assertions we wish to extract in our application.

After active sampling is complete, ListReader labels the text of each page as illustrated in Figure 3 and translates the labeled text into predicates and inserts them into the ontology. Record delimiter tags surround a complete record string and determine which fields belong to the same record. ListReader splits labels into object set names and instantiates objects for each new object set name and relationship predicates for each dot-separated sequence of object set names. The text string of the each leaf field is instantiated as a lexical object. Any remaining unlabeled text (text labeled as *NonList* or text still labeled with its original semantic ID) produces no output.

5. EVALUATION

We evaluate ListReader on two books, the *Shaver-Dougherty Genealogy* and the *Kilbarchan Parish Register*, and compare its performance to two baselines, an implementation of the Conditional Random Field (CRF) and a previous version of ListReader that induced regular-expression wrappers instead of HMM wrappers [Packer and Embley 2014]. The regex version of ListReader is similar to the HMM version except that it creates separate regular expression wrappers for every record pattern discovered during grammar induction whereas the HMM version is selective about which record and field group templates make it into the final HMM wrapper. The motivation for creating the HMM version is to overcome the brittleness of regular expressions, believing that the more malleable HMM wrappers would yield better recall results because of their ability to recognize variations in text patterns without requiring an exact match and would not hurt precision results too much because of ListReader's ability to create HMMs with a high degree of correlation to the observed text.

In Subsection 5.1, we describe the data (books) we used to evaluate ListReader. We explain the experimental procedure for evaluating the CRF in Subsection 5.2. We give the metrics we used in Subsection 5.3 and the results of the evaluation in Subsection 5.4, which includes a statistically significant improvement in F-measure as a function of labeling cost.

5.1. Data

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

We developed ListReader almost entirely using the text of the *The Ely Ancestry* [Beach et al. 1902] and *Shaver-Dougherty Genealogy* [Shaffer 1997]. *The Ely Ancestry* contains 830 pages and 572,645 word tokens and *Shaver-Dougherty Genealogy* contains 498 pages and 468,919 words. We used *Shaver-Dougherty Genealogy* and three pages of the *Kilbarchan Parish Register* [Grant 1912] containing 6013 words as our evaluation data. The *Kilbarchan Parish Register* would be considered a blind test except for our recognizing the need to not conflate lower-case words for this kind of book. We have added this option as an input parameter that is easy to set after quickly inspecting the input document. We chose the two test books to represent larger and more complex text on the one hand using the *Shaver-Dougherty Genealogy* and smaller and simpler text on the other using the *Kilbarchan Parish Register*.

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

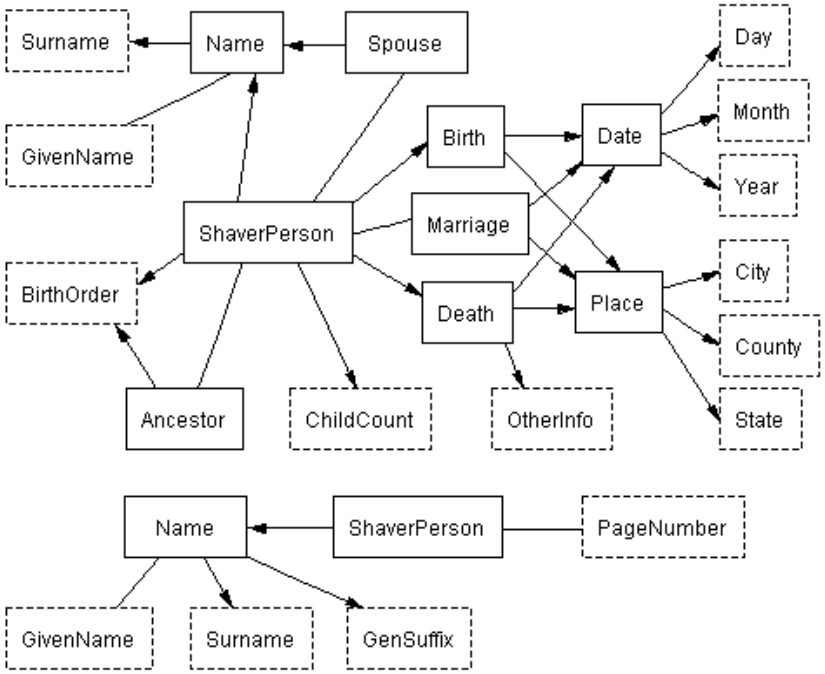


Fig. 16. ShaverPerson Ontologies

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

To label the text, we built a form in the ListReader web interface, like the one on the left side of Figure 1 that contains most of the information about a person visible in the lists of selected pages. Using the tool, we selected and labeled all the field strings in 68 pages from *Shaver-Dougherty Genealogy* and 3 pages from the *Kilbarchan Parish Register*. We ran the unsupervised wrapper induction on the text of the labeled pages. Grammar induction did not use the labels, but active sampling used a small number of them, namely those for the text selected by ListReader during active sampling. All of the remaining labels were used as ground truth for evaluation. The web form tool generated and populated the corresponding ontologies which we used as the source of labeled text. The annotated text from the 68 pages of the *Shaver-Dougherty Genealogy* have the following statistics: 14,314 labeled word tokens, 13,748 labeled field instances, 2,516 record instances, and 46 field types. Figure 16 shows the two ontologies used for these 46 field labels—one for the main body of the paper and one for the index. The annotated text from the 3 pages of the *Kilbarchan Parish Register* have the following statistics: 852 labeled word tokens, 768 labeled field instances, 165 record instances, and 12 field types. Figure 2 shows the ontology corresponding to those 12 field labels—the 12 paths from the *KilbarchanPerson* object set to the leaf lexical object sets after combining *MarriageDate* with *ProclamationDate* and combining *BirthDate* with *ChristeningDate*.

5.2. CRF Comparison System

We believe the performance of the supervised Conditional Random Field (CRF) serves as a good baseline or reference point for interpreting the performance of ListReader. The CRF implementation we applied is from the Mallet library [McCallum 2002]. To ensure a strong baseline, we performed feature engineering work to select an appropriate set of word token features that allowed the CRF to perform well on development test data. The features we applied to each word include the case-sensitive text of the word, and the following dictionary/regex Boolean attributes: given name dictionary (8,428 instances), surname dictionary (142,030 instances), names of months (25 variations), numeral regular expression, roman numeral regular expression, and name initial regular expression (a capital letter followed by a period). The name dictionaries are large and have good coverage of the names in the documents. We also distributed the full set of word features to the immediate left and right neighbors of each word token (after appending a "left neighbor" or "right neighbor" designation to the feature value) to provide the CRF with contextual clues. Using a larger neighbor window than just right and left neighbor did not improve its performance. These features constitute a greater amount of knowledge engineering than we allow for ListReader. We simulated active learning of a CRF using a random sampling strategy—considered to be a hard baseline to beat in active learning research, especially early in the learning process [Cawley 2011].

Each time we executed the CRF, we trained it on a random sample of n lines of text sampled throughout the hand-labeled portion of the corpus. Then we executed the trained CRF on all remaining hand-labeled text. We varied the value of n from 1 to 10 to fill in a complete learning curve. We ran the CRF 7,300 times for the *Shaver-Dougherty Genealogy* and 4,000 times for the *Kilbarchan Parish Register* and then computed the average y value (precision, recall, or F-measure) for each x value (cost) along the learning curve and generated a locally weighted regression curve from all 7,300 (or 4,000) points.

5.3. Experimental Procedure and Metrics

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

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

Since our aim is to develop a system that accurately extracts information at a low cost to the user, our evaluation centers on a standard metric in active learning research that combines both accuracy and cost into a single measurement: Area under the Learning Curve (ALC) [Cawley 2011]. The rationale is that there is no single, fixed level of cost that is right for all information extraction projects. Therefore, the ALC metric gives an average learning accuracy over many possible budgets. We primarily use F_1-measure as our measure of extraction accuracy, although we also report ALC for precision and recall curves. Precision is defined to be $\frac{tp}{tp+fp}$ and recall is defined to be $\frac{tp}{tp+fn}$ where *tp* means true positive, *fp* means false positive, and *fn* means false negative field strings. F-measure (F_1) is the harmonic mean of precision (p) and recall (r), or $\frac{2pr}{p+r}$. ALC is $\int_{min}^{max} f(c)dc$, where c is the number of user-labeled fields (cost) and $f(c)$ can be precision, recall, or F-measure as a function of cost, and *min* and *max* refer to the smallest and largest numbers of hand-labeled fields in the learning curve. The curve of interest for an extractor is the set of an extractor's accuracies plotted as a function of their respective costs. The ALC is the percentage of the area, between 0% and 100% accuracy and *min* and *max* cost, that is covered by the extractor's accuracy curve. ALC is equivalent to taking the mean of the accuracy metric at all points along the curve over the cost domain—an integral that is generally computed for discrete values using the Trapezoidal Rule,[5] which is how we compute it.

5.4. Results

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

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

Figures 17, 18, and 19 show plots of the F-measure, precision, and recall learning curves for ListReader and the CRF on the *Shaver-Dougherty Genealogy* and Figures 20, 21, and 22 show plots of the F-measure, precision, and recall learning curves for ListReader and the CRF on the *Kilbarchan Parish Register*. These plots provide detail behind the ALC metrics in Tables I and II. Visually, the learning curves indicate that ListReader (Regex and HMM) both outperform the CRF fairly consistently over varying numbers of field labels for all three metrics. Tables I and II tell us that the differences among the three extractors are statistically significant for most pairwise comparisons at $p < 0.05$ using an unpaired t test. The three pairs that are not signifi-

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

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

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

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

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

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

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

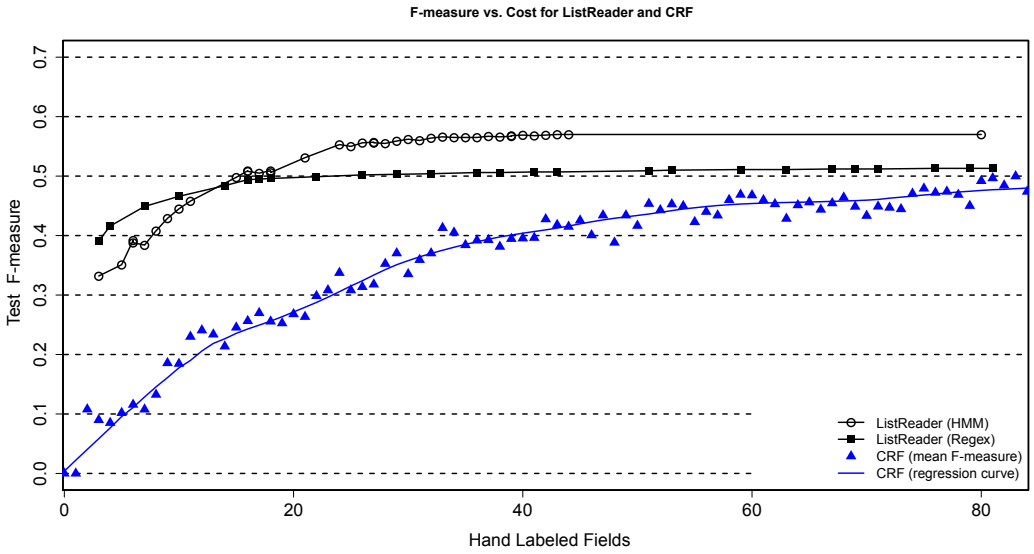


Fig. 17. F-measure Learning Curves for the *Shaver-Dougherty Genealogy*

cant are the ones comparing the recall of ListReader (Regex) and the CRF on both the *Shaver-Dougherty Genealogy* and the *Kilbarchan Parish Register* and comparing the precision of the two versions of ListReader on the *Kilbarchan Parish Register*.

The spike in the CRF's recall at Cost = 4 in Figure 22 is because the majority of records in the book are child records that contain 4 fields. When the CRF is lucky enough to train on one of these records, it usually does well extracting the other child record information.

Comparing the sizes of the extractors, ListReader (Regex) generated a regular expression that was 319,096 characters long for the *Shaver-Dougherty Genealogy* match-

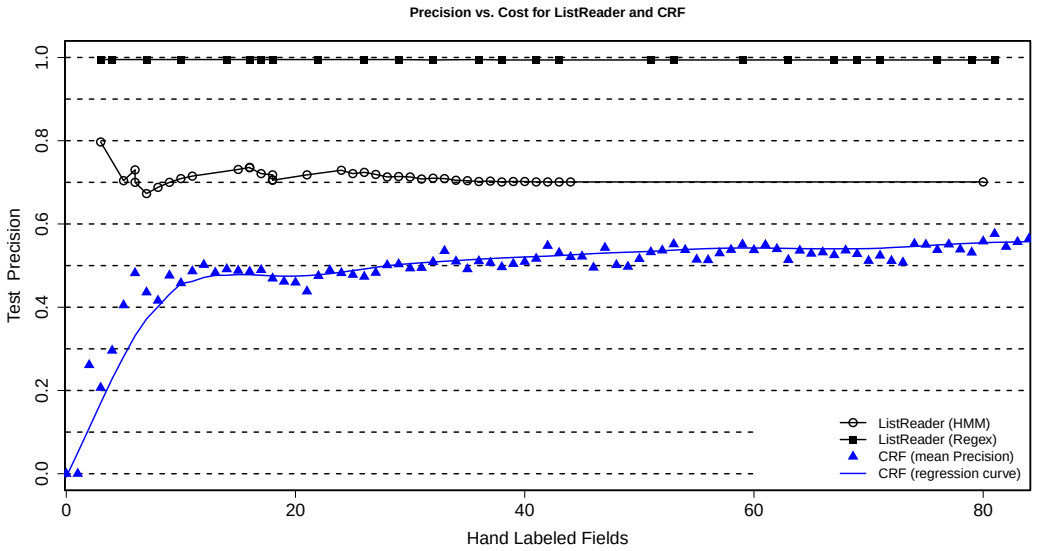


Fig. 18. Precision Learning Curves for the *Shaver-Dougherty Genealogy*

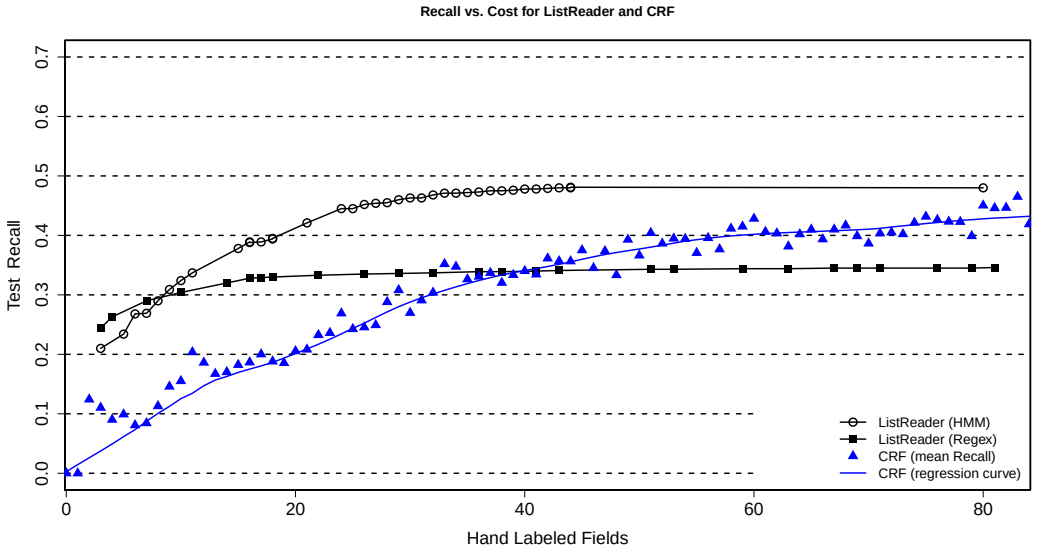


Fig. 19. Recall Learning Curves for the *Shaver-Dougherty Genealogy*

ing 3,334 records, and one that was 54,600 characters long for the *Kilbarchan Parish Register* matching 268 records. ListReader (HMM) generated an HMM with 2,015 states for the *Shaver-Dougherty Genealogy* matching 3,023 records and an HMM with 255 states for the *Kilbarchan Parish Register* matching 162 records. The HMM matches fewer records than the Regex because it is built from a fraction of the available record parse trees. The key to its improved recall, again, is that each (hand-labeled)

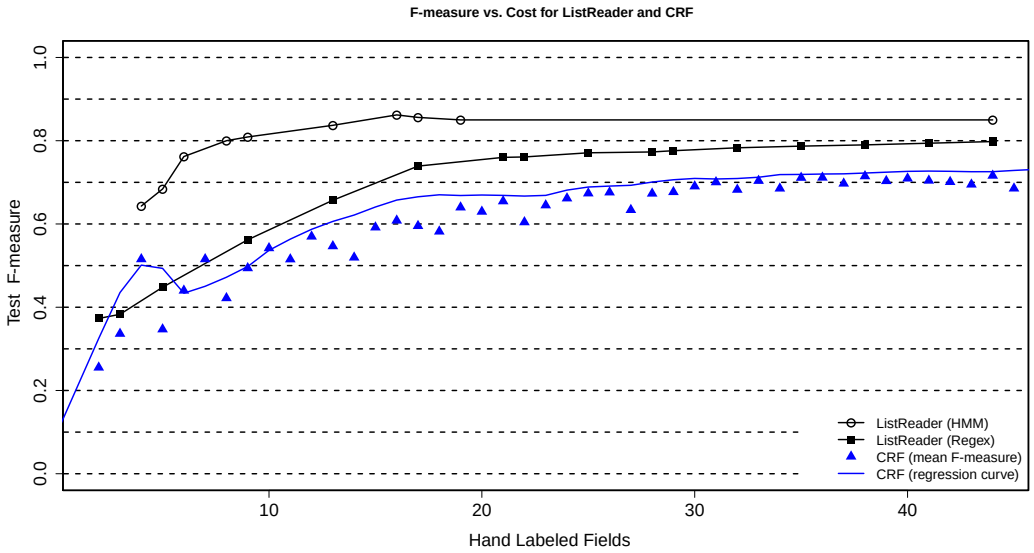


Fig. 20. F-measure Learning Curves for the *Kilbarchan Parish Register*

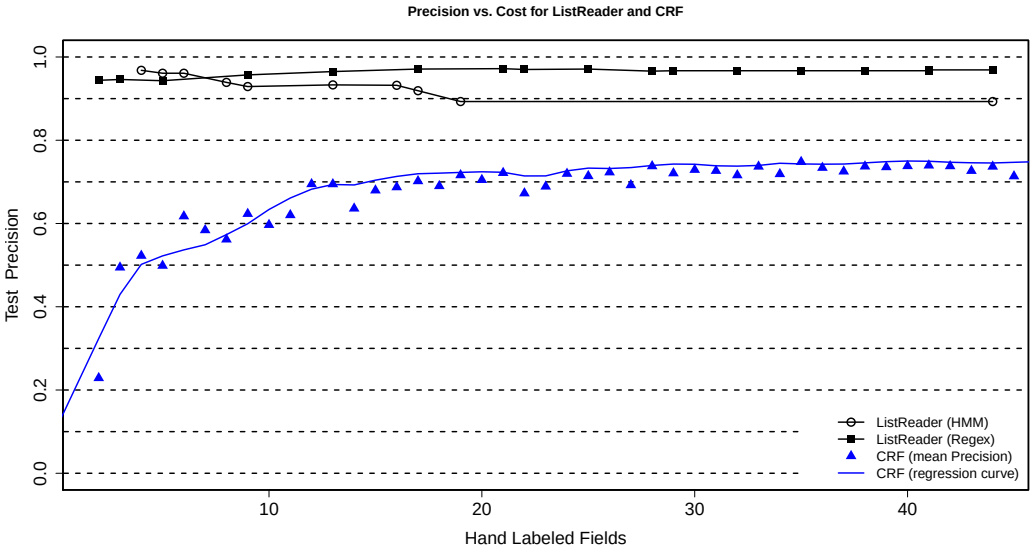


Fig. 21. Precision Learning Curves for the *Kilbarchan Parish Register*

HMM record template can match more records than each (hand-labeled) Regex template. Otherwise, the HMM should match less than half of the number of records that the Regex does. The CRF had 353 types of feature values and 28 states for the *Shaver-Dougherty Genealogy* and 191 types of feature values and 15 states for the *Kilbarchan Parish Register*. A smaller number of states probably contributed to its faster running time and lower accuracy compared to the HMM.

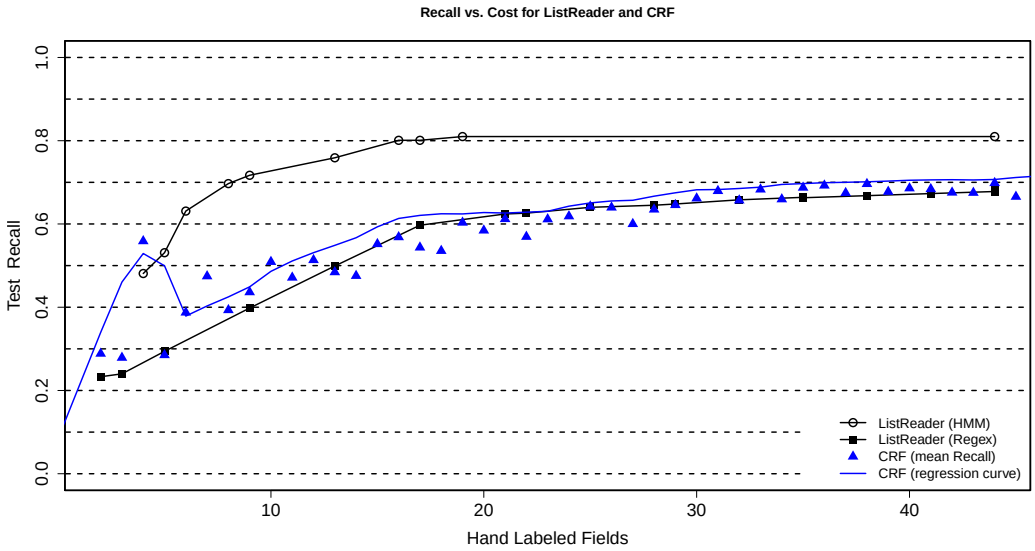


Fig. 22. Recall Learning Curves for the *Kilbarchan Parish Register*

Comparing the running time of ListReader, time and space complexity is linear in terms of the size of the input text, but unlike the Regex version, the HMM version is quadratic in the length of the record and the size of the label alphabet. The typical implementation of the training phase of a linear chain CRF is quadratic in both the sizes of the input text and the label set [Cohn 2007], [Guo et al. 2008]. We ran all extractors on a desktop computer with Java (JDK 1.7), a 2.39 GHz processor, and 3.25 GB of RAM. ListReader (Regex) took 26 seconds to run on the *Kilbarchan Parish Register* and 2 minutes 47 seconds to run on the *Shaver-Dougherty Genealogy*. ListReader (HMM) took 2 minutes 11 seconds to run on the *Kilbarchan Parish Register* and 59 minutes 18 seconds to run on *Shaver-Dougherty Genealogy*. The CRF took 9 seconds on *Kilbarchan Parish Register* and 52 seconds on *Shaver-Dougherty Genealogy*.

6. DISCUSSION AND FUTURE WORK

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

ListReader (HMM), as compared to ListReader (Regex), does relatively well in recall for the same reason it does relatively poorly in precision—by matching more text. By design, it uses only one "feature" per word token, and that feature is easily derived from the text, itself, without large knowledge resources. This is in contrast to our implementation of the CRF which, instead of removing information as our HMM does, the feature extractors add information. This makes the comparison CRF a less scalable

option in terms of development cost over multiple domains or text genres compared to our HMM whose main operating principle could be stated as "carefully throwing out the right kind of information." The technique of using semantic or lexical resources is somewhat more complicated in our work because of OCR errors that make dictionary matching more difficult. We thus do not currently rely on them.

On the other hand, adding semantic constraints to the HMM would likely help prevent some of its precision errors, such as labeling an "m." as a surname at the beginning of a line whose other text matched a known record pattern. Future work should investigate adding such semantic features or constraints to ListReader in a way that is cost-effective, for example using self-training, co-training, or bootstrapping that learns semantic categories from the input text, itself. We could also train ListReader from examples labeled automatically by other extractors, from wrappers trained on other books, or from examples that match a database of known facts such as the work in [Dalvi et al. 2010], with the added costs associated with those resources. Since the final mapping from HMM states to labels and predicates is the only step currently needing human labeled examples, adding a technique that utilizes automatically-labeled examples would make our approach completely unsupervised and scalable in terms of supervision cost.

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

7. RELATED WORK

Having described and evaluated ListReader, we now compare it with related research—wrapper induction in support of information extraction from semi-structured documents (Subsection 7.1) and unsupervised learning for extraction models (Subsection 7.2).

7.1. Automated Information Extraction from Lists

In general, related projects in the web wrapper induction literature [Chang et al. 2006] are almost universally applied to clean text—mostly to structured HMTL documents, and sometimes to semi-structured lists—allowing them to learn record patterns from as little as one page of input. The information extraction task of these systems is similar to ListReader's in that they extract, label, and group fields together that belong to the same record, and there can be multiple isolated records on a page. On the other hand, unlike ListReader, many wrapper induction approaches look for contiguous records, and those approaches that, like ListReader, are based on limited user input do not work with plain text (non-HTML) input and do not recognize multiple orderings of extracted fields. Those that work with plain text, like ListReader, use

syntactic and semantic features such as part of speech tags and WordNet categories, which ListReader does not need to use.

Choices in wrapper formalism include sets of left and right field context expressions ([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. Furthermore, none of these projects address all of the steps necessary to complete the process of the current research such as list finding, record segmentation, and field extraction.

The wrapper induction work most closely related to ListReader is IEPAD [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. The algorithmic complexity of IEPAD's wrapper construction phase appears to be quadratic because of its reliance on multiple string alignment, while ListReader's is linear. ListReader must use a different means of encoding (conflating) text than IEPAD so it can preserve more fine grained structure. IEPAD apparently cannot extract fields that are not explicitly delimited by some kind of HTML tag and looks only for contiguous records. Also, it appears that IEPAD users must identify pages containing target information; a ListReader user does not need to do so. IEPAD requires users to select patterns because the system may produce more than one pattern for a given type of record. ListReader automatically selects patterns among a set of alternatives. IEPAD users must also provide labels for each pattern, which is similar to the work of ListReader users, but is likely more difficult because it forces users to interpret induced patterns rather than original text, which raises the required user skill level. ListReader also minimizes the amount of supervision needed to extract a large volume of data by integrating an interactive labeling process into grammar induction, something IEPAD does not do. Lastly, ListReader reduces the cost of extracting information because of its unique combination of global pattern detection and active sampling—it focuses a user's effort on the most common patterns first, a valuable property not explored in any related information-extraction research.

7.2. Unsupervised Learning for Extraction Models

There are many wrapper induction projects applied to web pages that have a strong element of unsupervised machine learning, such as [Kushmerick 1997], [Ashish and Knoblock 1997], [Dalvi et al. 2010], and [Lerman et al. 2001]. These and other related research projects do not solve our targeted problem. Most do not address lists, specifically, and none address plain OCRed text. As Gupta and Sarawagi say ([Gupta and Sarawagi 2009]), the vast majority of methods of extraction of records from unstructured lists assume the presence of labeled unstructured records for training and a few assume a large database of structured records. None of these projects address all of the steps necessary to complete the process of the current research such as list finding, record segmentation, field extraction, and mapping to an expressive ontology.

A common and mathematically motivated means of unsupervised HMM induction is the Baum-Welch algorithm, an instance of the iterative Expectation-Maximization algorithm (EM). Baum-Welch finds the MLE parameters of an HMM in either unsu-

pervised or semi-supervised learning scenarios. In either case, text without manually-provided labels are assigned those labels that are most probable given the current HMM parameters, and those HMM parameters are in turn set from the most probable label distributions given the parameters set on the previous iteration. Grenager et al. ([Grenager et al. 2005]) use EM to train an HMM in both unsupervised and semi-supervised scenarios to extract fields from plain text records, including bibliographic citations and classified advertisements. They supplement EM with a few domain-dependent biases to prefer diagonal (self) transitions and recognize boundary tokens (punctuations). They report that the accuracy of the unsupervised approach starts low but is improved with the added biases. Furthermore, before adding the biases, their semi-supervised approach performed worse than supervised learning given the same number of hand-labeled examples, according to our reproduction of their work. The fields they extract are coarse-grained, such that a sequence of author names in a bibliographic citation is considered one homogeneous segment. Our work differs from theirs in that we set the HMM parameters from record structure proposed by a separate phrase grammar that we induce automatically and separately (without any connection to the HMM). We also extract more fine-grained information, e.g. individual person names and parts of those names, to improve the richness of the resulting data.[6] Therefore, their self-transition bias would not be appropriate in our work. Also, Grenager et al. assume that list records have been found and extracted before their process begins, which we do not assume for ours. Unlike the semi-supervised part of their work, we do not perform any training of the HMM's structure or parameters using hand labeled data which may be a more scalable approach given a large input corpus.

Elmeleegy et al. ([Elmeleegy et al. 2009]) present an algorithm to automatically convert a source HTML list into a table, with no hand-labeled training data and no output labeling of fields or columns. They segment fields in records automatically using the following sources of information to predict which words should be split and which should remain together: (1) sets of "data type" regular expressions including common numeric entity patterns, (2) an n-gram language model producing internal cohesiveness and external in-cohesiveness scores, and (3) a thresholded count of the number of cells matched in a corpus of extracted table cells. They combine these sources of evidence using a weighted average. They also correct errors in the first pass of segmentation by counting fields, forcing all records to be segmented into no more than the most common number of fields, and aligning shorter records using a modification of the Needleman-Wunsch algorithm. Like Grenager et al., they perform field segmentation and alignment but do not appear to perform list discovery or record segmentation as we do. They also do not label fields or fully extract information, and they target HTML lists which may contain additional formatting clues not present in our OCR text. Unlike us, they assume that the order of fields does not change between list entries. Unsupervised techniques like theirs target web-scale applications and they also rely on a web-scale corpus. Therefore, they avoid hand-labeling of training data. Their source table data is a massive collection of tables from the web. Using massive amounts of web data is a common technique among recent web wrapper research that rely on the sheer size of the web as a key resource for their system. We do not use web-scale data resources. They assume there are not many optional fields in their input data which is not true of our data. Forcing the number of fields/columns to equal the mode of the

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

field count per row discovered in the first pass will not work correctly for many lists because there can be optional fields which do not often occur.

Gupta and Sarawagi ([Gupta and Sarawagi 2009]) convert HTML source lists on the web into tables that match and augment an incomplete user-provided table. Their unsupervised approach first ranks lists with a Lucene query, based on the words in the user-provided table. Second, they label candidate fields in the source list records as training data by marking text in the list records that match text in the columns of the user-provided table. Third, they train a separate CRF for each source list using the automatically labeled records of the list and then apply the CRF to the rest of the records of that list. This effectively produces tables from the lists. They finally merge and rank the rows of the resulting tables and returns the top ranked rows of the final table to the user. Rows that repeat often in source lists and which are given high confidence scores by the CRF are ranked high. This work is similar to ours in that they train a statistical sequence model on the text of lists labeled by a separate, automatic process. It differs from ours in that their source text (web pages) have no OCR errors and have more structure making it easier to find lists, segment records, and identify fields. They do not need to complete a mapping from text fields to ontology predicates, they only need to align user-provided fields with fields in a list record. They do not seem to (or need to) segment records in lists before extracting fields. They have a much larger source of potential lists than we do and only need to find some with high accuracy, not all of them. In our project, we evaluate against an ideal of extracting all list records from a book. This work, as well as the other two, do not extract richly- and explicitly-structured data suitable for ontology population as we do.

8. CONCLUDING REMARKS

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

APPENDIX: Example Pages

For reference, we append an example page from *The Ely Ancestry* in Figure 23, from the *Shaver-Dougherty Genealogy* in Figure 24, and three pages from the *Kilbarchan Parish Register* in Figures 25, 26, and 27.

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.

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

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

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

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

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

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

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

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

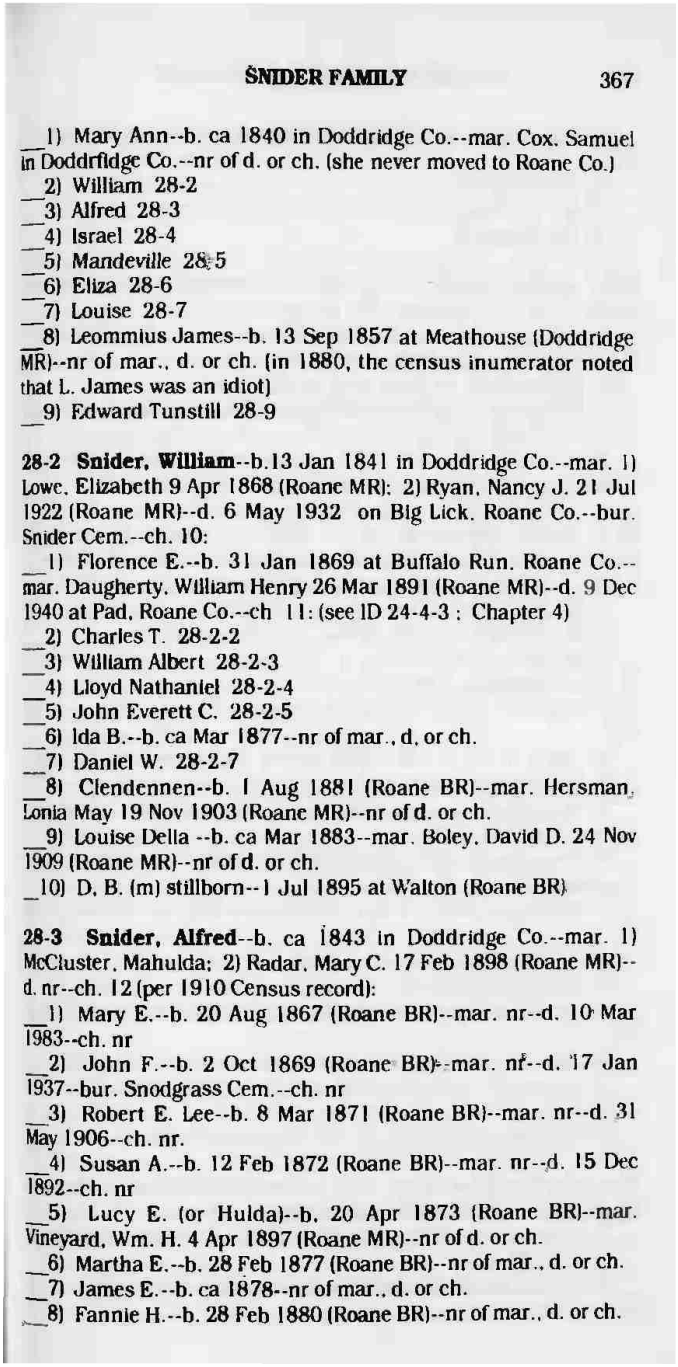
The attempt was gallantly made, but was only partially successful, owing to the ship's thick copper sheathing.

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

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

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

Fig. 23. Page from *The Ely Ancestry*, Page 367



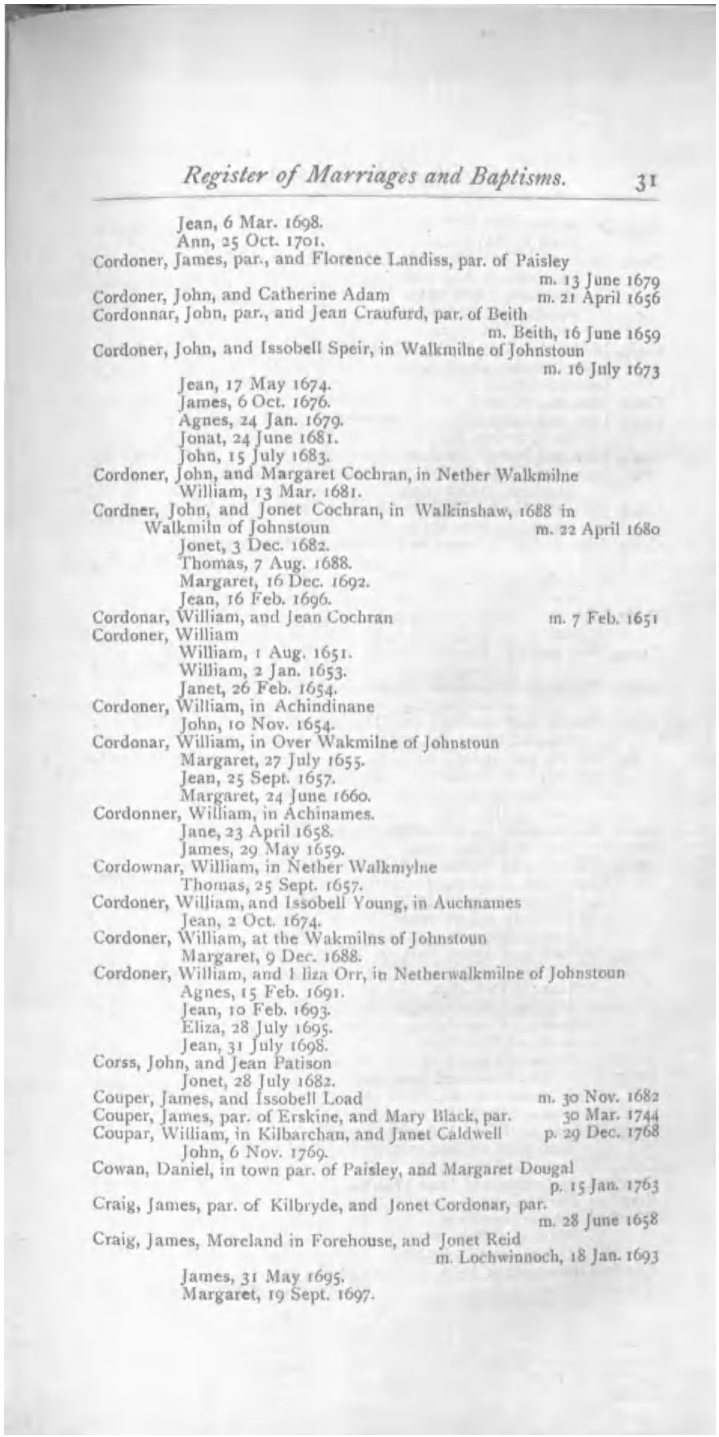
SNIDER FAMILY 367

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

28-2 **Snider, William**--b.13 Jan 1841 in Doddridge Co.--mar. 1) Lowe, Elizabeth 9 Apr 1868 (Roane MR); 2) Ryan, Nancy J. 21 Jul 1922 (Roane MR)--d. 6 May 1932 on Big Lick, Roane Co.--bur. Snider Cem.--ch. 10:
__1) Florence E.--b. 31 Jan 1869 at Buffalo Run, Roane Co.--mar. Daugherty, William Henry 26 Mar 1891 (Roane MR)--d. 9 Dec 1940 at Pad, Roane Co.--ch 11: (see ID 24-4-3 ; Chapter 4)
__2) Charles T. 28-2-2
__3) William Albert 28-2-3
__4) Lloyd Nathaniel 28-2-4
__5) John Everett C. 28-2-5
__6) Ida B.--b. ca Mar 1877--nr of mar., d, or ch.
__7) Daniel W. 28-2-7
__8) Clendennen--b. 1 Aug 1881 (Roane BR)--mar. Hersman, Lonia May 19 Nov 1903 (Roane MR)--nr of d. or ch.
__9) Louise Della --b. ca Mar 1883--mar. Boley, David D. 24 Nov 1909 (Roane MR)--nr of d. or ch.
_10) D, B. (m) stillborn--1 Jul 1895 at Walton (Roane BR)

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

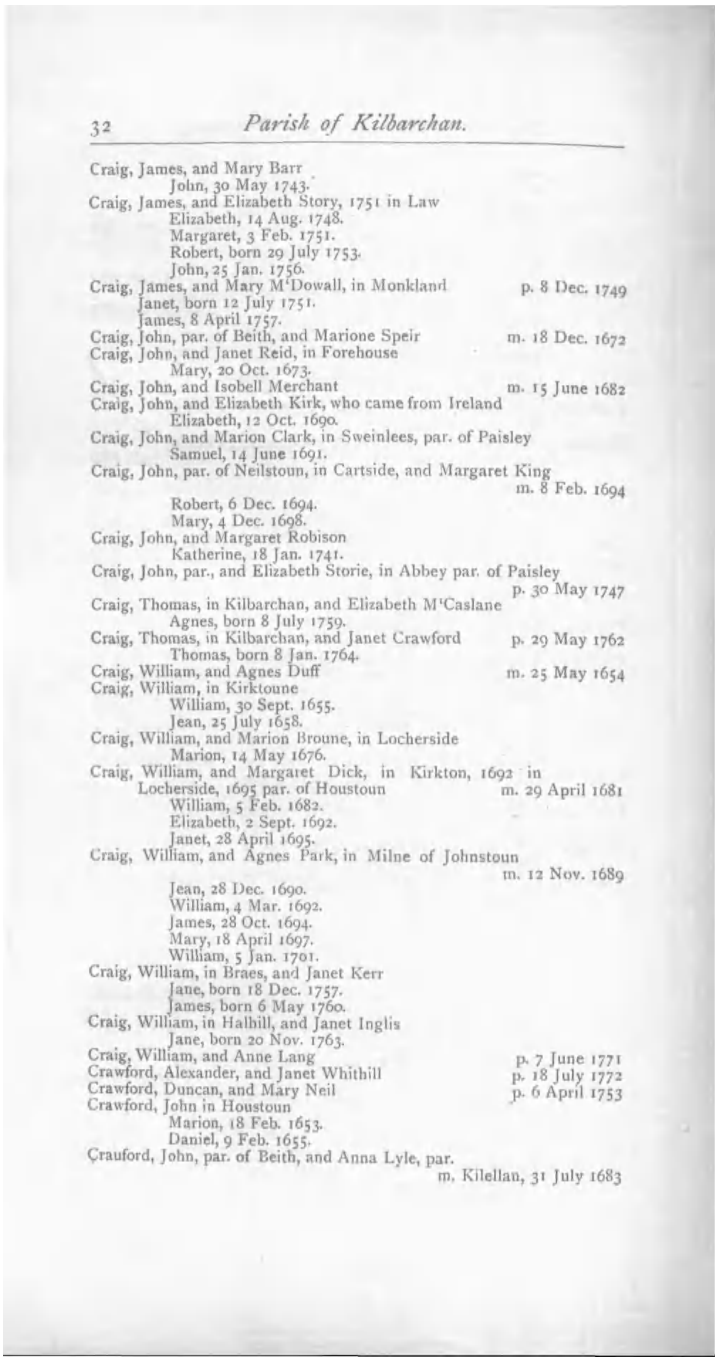
Fig. 24. Page from *Shaver-Dougherty Genealogy*, Page 154



Register of Marriages and Baptisms. 31

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

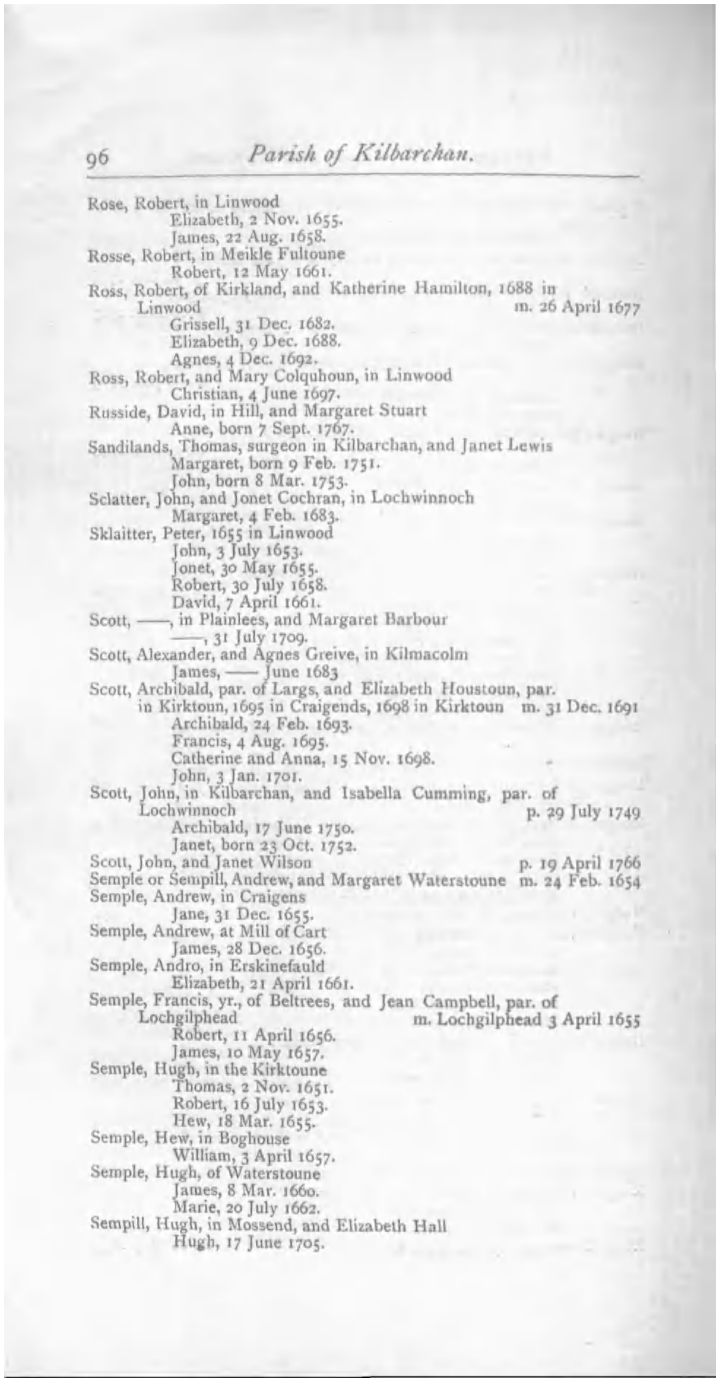
Fig. 25. Page from *Kilbarchan Parish Register*, Page 31



32 *Parish of Kilbarchan.*

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

Fig. 26. Page from *Kilbarchan Parish Register*, Page 32



96 *Parish of Kilbarchan.*

Rose, Robert, in Linwood
Elizabeth, 2 Nov. 1655.
James, 22 Aug. 1658.
Rosse, Robert, in Meikle Fultoune
Robert, 12 May 1661.
Ross, Robert, of Kirkland, and Katherine Hamilton, 1688 in Linwood m. 26 April 1677
Grissell, 31 Dec. 1682.
Elizabeth, 9 Dec. 1688.
Agnes, 4 Dec. 1692.
Ross, Robert, and Mary Colquhoun, in Linwood
Christian, 4 June 1697.
Russide, David, in Hill, and Margaret Stuart
Anne, born 7 Sept. 1767.
Sandilands, Thomas, surgeon in Kilbarchan, and Janet Lewis
Margaret, born 9 Feb. 1751.
John, born 8 Mar. 1753.
Sclatter, John, and Jonet Cochran, in Lochwinnoch
Margaret, 4 Feb. 1683.
Sklaitter, Peter, 1655 in Linwood
John, 3 July 1653.
Jonet, 30 May 1655.
Robert, 30 July 1658.
David, 7 April 1661.
Scott, ——, in Plainlees, and Margaret Barbour
——, 31 July 1709.
Scott, Alexander, and Agnes Greive, in Kilmacolm
James, —— June 1683
Scott, Archibald, par. of Largs, and Elizabeth Houstoun, par. in Kirktoun, 1695 in Craigends, 1698 in Kirktoun m. 31 Dec. 1691
Archibald, 24 Feb. 1693.
Francis, 4 Aug. 1695.
Catherine and Anna, 15 Nov. 1698.
John, 3 Jan. 1701.
Scott, John, in Kilbarchan, and Isabella Cumming, par. of Lochwinnoch p. 29 July 1749
Archibald, 17 June 1730.
Janet, born 23 Oct. 1752.
Scott, John, and Janet Wilson p. 19 April 1766
Semple or Sempill, Andrew, and Margaret Waterstoune m. 24 Feb. 1654
Semple, Andrew, in Craigens
Jane, 31 Dec. 1655.
Semple, Andrew, at Mill of Cart
James, 28 Dec. 1656.
Semple, Andro, in Erskinefauld
Elizabeth, 21 April 1661.
Semple, Francis, yr., of Beltrees, and Jean Campbell, par. of Lochgilphead m. Lochgilphead 3 April 1655
Robert, 11 April 1656.
James, 10 May 1657.
Semple, Hugh, in the Kirktoune
Thomas, 2 Nov. 1651.
Robert, 16 July 1653.
Hew, 18 Mar. 1655.
Semple, Hew, in Boghouse
William, 3 April 1657.
Semple, Hugh, of Waterstoune
James, 8 Mar. 1660.
Marie, 20 July 1662.
Sempill, Hugh, in Mossend, and Elizabeth Hall
Hugh, 17 June 1705.

Fig. 27. Page from *Kilbarchan Parish Register*, Page 96

REFERENCES

Naveen Ashish and Craig 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.

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.

Chia Hui Chang, Mohammed Kayed, M.R. Girgis, and K.F. Shaalan. 2006. A Survey of Web Information Extraction Systems. *IEEE Transactions on Knowledge and Data Engineering* 18, 10 (Oct. 2006), 1411–1428. DOI:http://dx.doi.org/10.1109/TKDE.2006.152

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.

Charles Elkan and Keith Noto. 2008. Learning Classifiers from Only Positive and Unlabeled Data. In *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '08)*. ACM, New York, NY, USA, 213–220. DOI:http://dx.doi.org/10.1145/1401890.1401920

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.

Francis J. Grant (Ed.). 1912. *Index to the Register of Marriages and Baptisms in the Parish of Kilbarchan, 1649 - 1772*. J. Skinner and Company, Ltd., Edinburgh, Scotland.

Trond Grenager, Dan Klein, and Christopher D. Manning. 2005. Unsupervised Learning of Field Segmentation Models for Information Extraction. In *Proceedings of the Forty-third Annual Meeting on Association for Computational Linguistics*. Ann Arbor, Michigan, USA, 371–378.

Yong Zhen Guo, Kotagiri Ramamohanarao, and Laurence A. F. 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.

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

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

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

Yanliang Li, Jing Jiang, Hai Leong Chieu, and Kian Ming A. Chai. 2011. Extracting Relation Descriptors with Conditional Random Fields. *Proceedings of the 5th International Joint Conference on Natural Language Processing* (2011), 392–400.

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

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

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*. Technical Report. Department of Computer Science, Brigham Young University, Provo, Utah. 1–48 pages. http://deg.byu.edu/papers (Submitted to TKDD).

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.

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

Received Month 0000; revised Month 0000; accepted Month 0000