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 learn a mapping from facts asserted in text to structured facts asserted in an ontology. In this paper, we propose 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 associating subsets of identical fields across related 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 (HMM) learns faster 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), 35 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. 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 number of manually selected and annotated examples in order to learn.

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

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

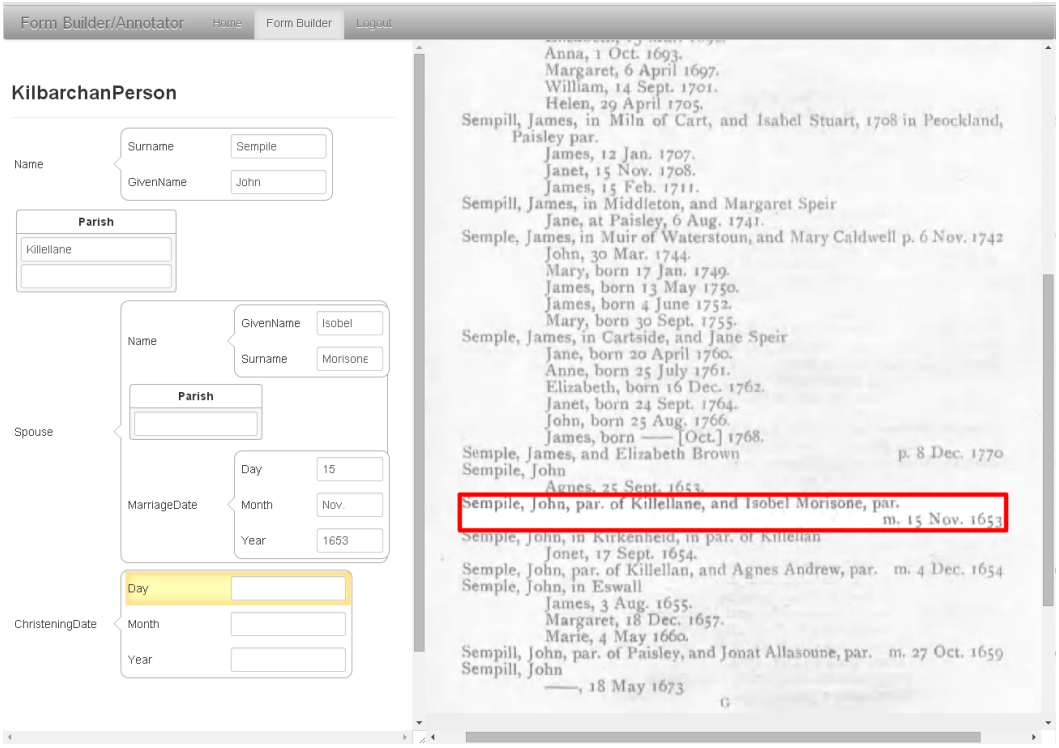


Fig. 1. KilbarchanPerson Page and Filled-in Form

We propose *ListReader*, an unsupervised active wrapper induction process for learning Hidden Markov Models (HMMs) 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 an ontology. In the end, ListReader induces a wrapper that is at least as accurate as a typical 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 the process, 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 targeted for an information application. For example, the user could select the *Kilbarchan Parish Register* [Grant 1912], part of one page of which 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 in Figure 1 before being filled in.[1] ListReader translates the form into an ontology schema, e.g. the empty target ontology in Figure 2. Without anything more than the given text document (e.g. the entire collection of page images

[1]The construction of a form is the full extent of human manual knowledge engineering for the entire wrapper induction process.

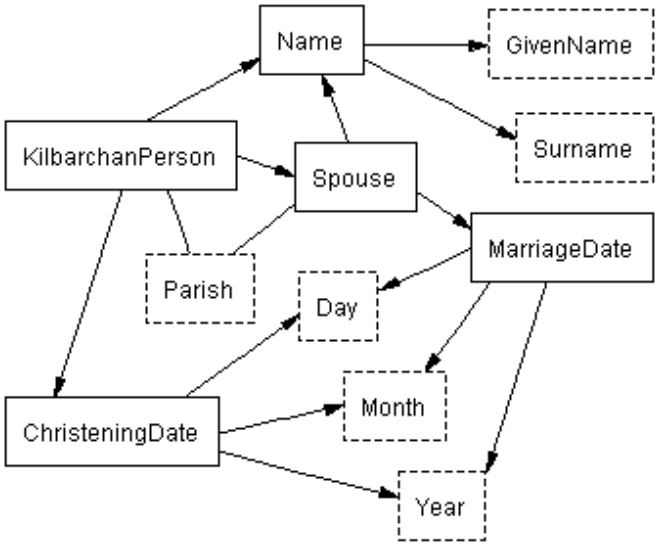


Fig. 2. KilbarchanPerson Ontology

```
<KilbarchanPerson.Name.Surname>Sempile</KilbarchanPerson.Name.Surname>,
<KilbarchanPerson.Name.GivenName>John</KilbarchanPerson.Name.GivenName>, par. of
<KilbarchanPerson.Parish[1]>Killellane</KilbarchanPerson.Parish>[1], and
<KilbarchanPerson.Spouse.Name.GivenName>Isobel</KilbarchanPerson.Spouse.Name.GivenName>
<KilbarchanPerson.Spouse.Name.Surname>Morisone</KilbarchanPerson.Spouse.Name.Surname>,
par. m. <KilbarchanPerson.Spouse.MarriageDate.Day>15</KilbarchanPerson.Spouse.MarriageDate.Day>
<KilbarchanPerson.Spouse.MarriageDate.Month>Nov</KilbarchanPerson.Spouse.MarriageDate.Month>.
<KilbarchanPerson.Spouse.MarriageDate.Year>1653</KilbarchanPerson.Spouse.MarriageDate.Year>
```

Fig. 3. Labeled Record "Sempile, John, par. of Killellane, and Isobel Morisone, par. m. 15 Nov. 1653"

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 wishes the user to label by highlighting them, as Figure 1 indicates. 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 in the highlighted record. After labeling, the structure and parameters of the HMM are unchanged but some of the states 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

[2]Filling in the form from ListReader selected text is the full extent of hand labeling for the entire wrapper induction process.

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 be applied 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.

The contributions of this research are the following. 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 and the discovered pattern length. Second, we provide an efficient way to complete the HMM as a data-extraction wrapper that can map the data in lists to an expressive ontology schema. The final wrapper outperforms two alternatives.

We give the details of these contributions as follows. Sections 2, 3, and 4 explain the HMM wrapper induction process. Section 2 describes how ListReader discovers record-like patterns in text in linear time and space without human input. Section 3 describes how ListReader derives the structure and parameters of an HMM from the discovered patterns, also without human supervision. Section 4 explains how ListReader creates the mapping from HMM states to an ontology using active sampling, an active-learning-like process that requests labels of select examples from the user without modifying the internal wrapper structure. Section 5 provides an evaluation of the performance of ListReader in terms of the precision, recall, and F-measure of the automatically extracted information as a function of manual field labeling cost, and compares the learning rates to a state-of-the-art statistical sequence labeler (CRF) and to the previous version of ListReader that induces regular expression based wrappers. Section 6 discusses performance issues and opportunities for future work. Section 7 compares our proposed solution to related work on unsupervised wrapper induction. Finally, Section 8 concludes this paper.

2. UNSUPERVISED PATTERN DISCOVERY

In an unsupervised process of pattern discovery, ListReader finds record-like patterns in the input text, provides a representation of the hierarchical structure of these strings, and associates the major components (delimited field groups) across different types of records. In Section 3, ListReader will flatten this hierarchical structure into a state machine and set the parameters of the HMM using statistics in the collection of parsed record patterns. As we explain in Subsection 2.1, ListReader begins parsing to discover patterns by conflating the input text—substituting abstract word and phrase structure for strings of characters. ListReader then efficiently identifies record-like patterns in the conflated 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 ListReader will construct 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.

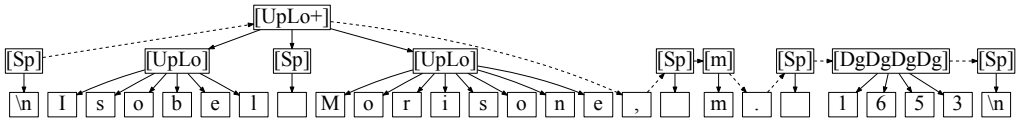


Fig. 4. Initial Parse Tree of "`\nIsobel Morisone, m. 1653\n`"

(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 horizontally-oriented punctuation characters: underscore, hyphen, en dash, em dash, and other Unicode variations.
(3) *Numeral*: Replaces each digit in a numeral with a generic digit symbol ("`Dg`").
(4) *Word*: Replaces contiguous letters with a generic word symbol that preserves the relative order of upper case ("`Up`") and lower case ("`Lo`") characters only. 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 sequences of space-delimited, capitalized words of any length with a single, generic symbol ("`[UpLo+]`").
(8) *Numeral Repetition*: Replaces sequences 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.

The cumulative application of these rules produces a sequence of roots of small parse trees. Figure 4 shows the parse tree for the text "`\nIsobel Morisone, m. 1653\n`". The dashed line joins the sequence of root phrase symbols giving a new sequence of symbols in which ListReader looks for patterns. Not all rules need be used for every book. Generally, all of them should be used except when they 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 fairly well structured.

2.2. Record Pattern Search

Once the text is parsed, ListReader finds record-like patterns in it by building a suffix tree data structure from the conflated text and searching for repeated patterns that satisfy our record selection constraints including the following: records must begin and end with a valid record delimiter character, must occur a minimum number of times throughout the text (two or three times, depending on the size of the text), must contain at least one numeral or capitalized word.

Figure 5 shows the suffix tree built from two consecutive and simplified records[3] of the form "`[Sp][UpLo+],[Sp][m].[Sp][DgDgDgDg][Sp]`", e.g. "`\nIsobel Morisone, m. 1653\nJonat Allasoune, m. 1659\n`", that share the

[3]We use this artificial example to make the suffix tree small enough to discuss in a paper. Normally, suffix trees are much larger because they represent every suffix of the input text.

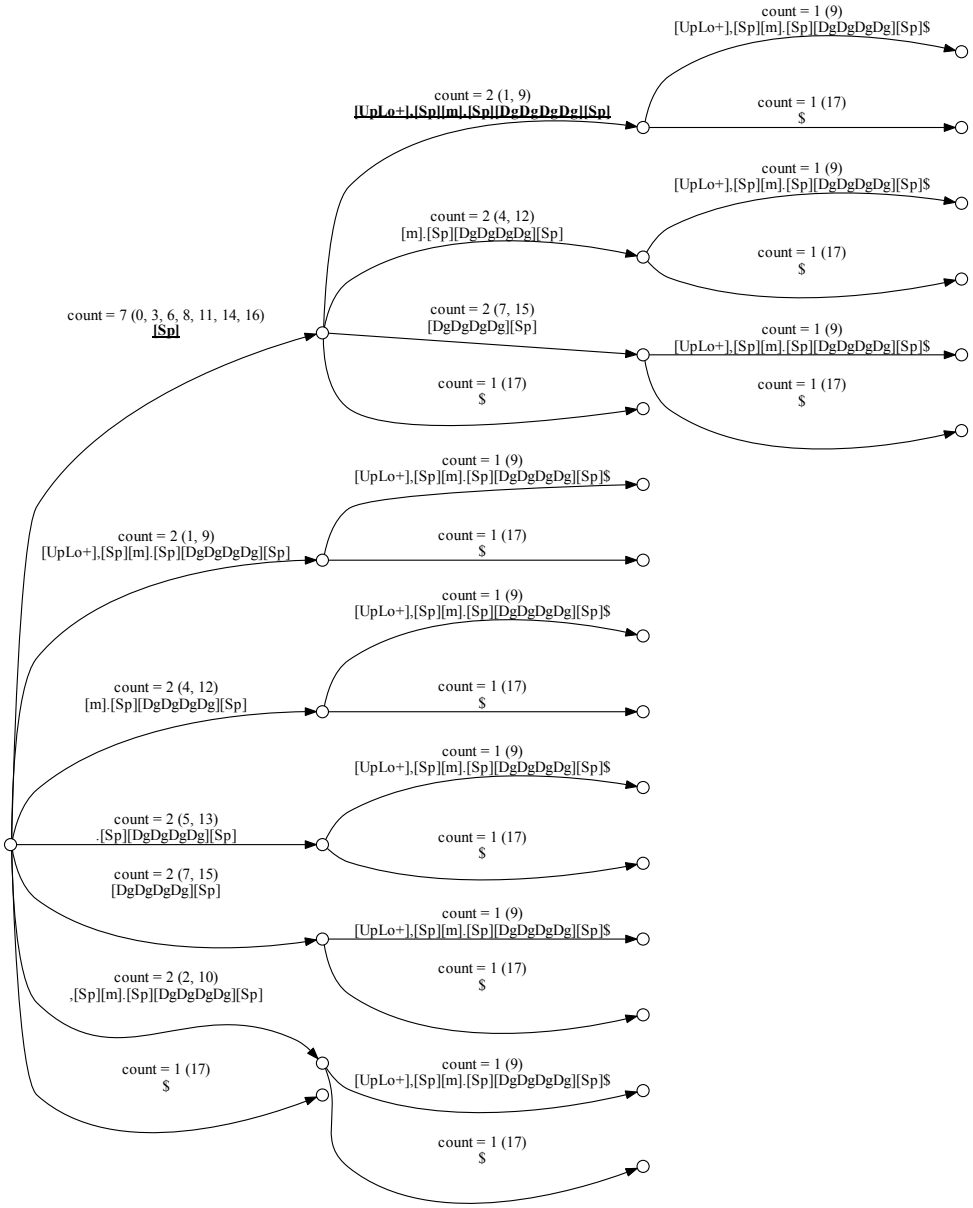


Fig. 5. Suffix Tree of "`[Sp] [UpLo+] , [Sp] [m] . [Sp] [DgDgDgDg] [Sp] [UpLo+] , [Sp] [m] . [Sp] [DgDgDgDg] [Sp]`".

sequence of symbols in Figure 4. ListReader constructs the 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

1.	[Sp][UpLo+],[Sp][UpLo+],[Sp][in][Sp][UpLo+],[Sp][par].[Sp][of][Sp][UpLo+], [Sp][and][Sp][UpLo+][Sp]
	\nSteel, John, in Peockland, par. of Paisley, and Betbiah\n
	\nSempill, John, in Burneight, par. of Lochwinnoch, and Janet Cochrane\n
2.	[Sp][UpLo+],[Sp][UpLo+],[Sp][par].[Sp][of][Sp][UpLo+], [Sp][and][Sp][UpLo+],[Sp][par].[Sp][m].[Sp][DgDg][Sp][UpLo+].[Sp][DgDgDgDg][Sp]
	\nSempile, John, par. of Killellane, and Isobel Morisone, par.\nm. 15 Nov. 1653\n
	\nSempill, John, par. of Paisley, and Jonat Allasoune, par. m. 27 Oct. 1659\n
3.	[Sp][UpLo+],[Sp][DgDg][Sp][UpLo+].[Sp][DgDgDgDg][Sp]
	\nAgnes, 25 Sept. 1653.\n
	\nJonet, 17 Sept. 1654.\n
	\nMargaret, 18 Dec. 1657.\n
4.	[Sp][UpLo+],[Sp][Dg][Sp][UpLo+][Sp][DgDgDgDg][Sp]
	\nWilliam, 6 June 1690.\n
	\nAndrew, 6 May 1692.\n
	\nMarie, 4 May 1660.\n
5.	[Sp][UpLo+],[Sp][UpLo+][Sp]
	\nCordoner, William\n
	\nRose, Robert\n

Fig. 6. A Selection of Patterns Found in the *Kilbarchan Parish Register*

adhere to the properties of a record. The one pattern in Figure 5 that ListReader would find is underlined and is represented by the concatentation of two edges. Notice that this pattern begins and ends with space symbols which are connected to newline characters in the parse tree. The pattern also repeats 2 times (given the count of the second edge).

Figure 6 shows five other patterns that would have been discovered in a suffix tree constructed from a larger sample of the text from the *Kilbarchan Parish Register*. Each record pattern is connected to a set of aligned record candidate strings in the input text forming a record cluster. Figure 6 shows a sample of 12 records grouped in five record clusters.

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.

From the set of record clusters discovered, ListReader identifies field group delimiters: sequences of lower-case words separated by whitespace or punctuation that occur in a fixed position within a few different record clusters (between two and four clusters, depending on the size and complexity of the input text). Requiring more than four record clusters typically eliminates valid delimiters from consideration, and requiring less than two record clusters provides insufficient evidence for delimiter patterns. To find field group delimiters, ListReader compares the original text of each word at each position within a record template (below the root nodes in the parse trees). When it finds delimiters to add to a field group template, it inserts the non-conflated text instead of the conflation symbols. From the first two clusters in Figure 6, ListReader can identify the following field group delimiters: “, par. of ”, “, and ”, and “\n”. With additional supporting evidence from other clusters in the book, ListReader might also identify the other two delimiters seen in the first two record clusters in Figure 6: “, in ” and “, par. m. ”.

ListReader constructs field group templates from the text appearing between field group delimiters and associates the field group template with the delimiter on its left side. Field group templates consist of one or more variations of a field group delimiter,

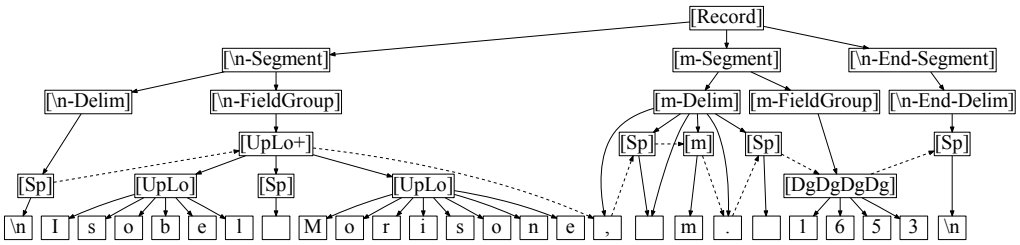


Fig. 7. Complete Parse Tree of "`\nIsobel Morisone, m. 1653\n`"

which text *is not* conflated, followed by one or more variations of the field group, which text *is* conflated. For example, the field group template for the delimiter "`, par. of `" discovered in the first two record clusters in Figure 6 would be "`, par. of [UpLo+]`" and for the delimiter "`, and `" would be "`, and [UpLo+]`". When applied to parse text, we say that a field group template produces a "field group segment" as a new type of parse tree node. These can be seen in parse tree in Figure 7 where each "Segment" node includes a "Delim" node followed by a "FieldGroup" node. A special "End-Segment" node that includes a single "End-Delim" node marks the end record delimiter as Figure 7 shows.

2.4. Final Record and Field Group Template Selection

As a next step, ListReader again produces a suffix tree, working at the level of field-group segments. This more coarsely-grained text string at the level of field-group segments for the parse tree in Figure 7, for example, is "`[\n-Segment][m-Segment][\n-End-Segment]`". For two such consecutive segments,[4] ListReader produces the suffix tree in Figure 8. ListReader then searches for record patterns in the second suffix tree as it did in the first suffix tree with one additional constraint: each record template must be composed entirely of field group segments. (Any record template that, for example, has an extra punctuation character that was not incorporated into any field group segment is not considered.) This produces a higher level of text abstraction. Figure 9 shows how the records from Figure 6 would now appear after parsing field group segments and rediscovering records. Notice how the third, fourth, and fifth record clusters are now merged because they share a common segment-level parse sequence, namely, the initial `[\n-Segment]`. These selected record templates define the sets of records from which ListReader later constructs the HMM, and from which the HMM's parameters will be set.

For each of these record templates, all of the members of its record cluster share the same conflation pattern. ListReader is therefore guaranteed that the same type of field group segment appears in the same position within all of its record instances. For example, the two records in the first cluster in Figure 9 all contain the following sequence of field group segments: an initial "`\n`" segment, an "`in`" segment, a "`par-of`" segment, an "`and`" segment, and a final "`\n`" segment. On the other hand, the specific contents of the field group segments may differ from each other even though they are aligned within the same kind of record template. For example, the first record cluster in Figure 9 contains spouse names that are of different length. Greater variations than this

[4]Again, we mention the use of an artificial example to make the suffix tree small enough to discuss in a paper. Normally, hundreds and even thousands of occurrences of a record-segment pattern are found scattered throughout the document.

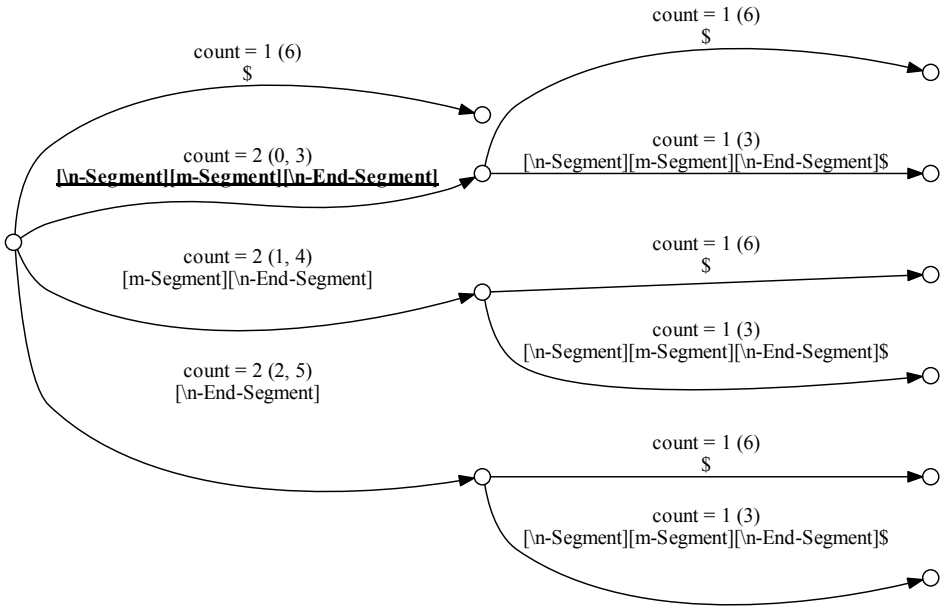


Fig. 8. Phase 2 suffix tree of "`[\n-Segment][m-Segment][\n-End-Segment][\n-Segment][m-Segment][\n-End-Segment]`". The record pattern is underlined.

1.	`[\n-Segment][in-Segment][par-of-Segment][and-Segment][\n-End-Segment]`
	`\nSteel, John, in Peockland, par. of Paisley, and Betbiah\n`
	`\nSempill, John, in Burneight, par. of Lochwinnoch, and Janet Cochrane\n`
2.	`[\n-Segment][par-of-Segment][and-Segment][par-m-Segment][\n-End-Segment]`
	`\nSempile, John, par. of Killellane, and Isobel Morisone, par.\nm. 15 Nov. 1653\n`
	`\nSempill, John, par. of Paisley, and Jonat Allasoune, par. m. 27 Oct. 1659\n`
3.	`[\n-Segment][\n-End-Segment]`
	`\nAgnes, 25 Sept. 1653.\n`
	`\nJonet, 17 Sept. 1654.\n`
	`\nMargaret, 18 Dec. 1657.\n`
	`\nWilliam, 6 June 1690.\n`
	`\nAndrew, 6 May 1692.\n`
	`\nMarie, 4 May 1660.\n`
	`\nCordoner, William\n`
	`\nRose, Robert\n`

Fig. 9. Segment-level Record Clusters Derived from the Record Clusters in Figure 6

are common, such as aligning "`, par.\nm. 15 Nov. 1653\n`" with "`, par. m. 1653\n`". And, even greater variations shown in the third record cluster in Figure 9 where the start of a record can be either a name/comma/date or a surname/comma/given-name. Because of this, ListReader must either create a separate sub-HMM for each variation, select among these variations and create sub-HMMs for only a subset of the variations, or merge the variations into some kind of union of sub-HMMs. Using all possible variations is less desirable for two reasons: it significantly increases the run-

ning 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 each field group variation at each position of each record template.

For each field group segment in each record template, ListReader selects one or more field group representatives. For "`[\n-Segment]`" in the third segment-level record cluster in Figure 9, for example, ListReader selects one or more of the corresponding patterns that comprise it—one or more of the corresponding patterns from the third, fourth, and fifth patterns in Figure 6. Ideally, the representatives for a segment together cover each major variation for the segment. To select the representatives for a segment, ListReader first selects one of the alternative patterns at random. ListReader then iterates over the remaining alternatives in random order. For each field group template variation, ListReader computes its normalized Levenstein edit distance from each of the representative templates among those being considered. The field group template must have an edit distance of 0.25 or less to be grouped together in the same set. Having a normalized edit distance of 0.25 means that the field group template perfectly overlaps with 75% of the words of the representative of that set—the first one chosen randomly to initialize the set. (Delimiter words match if they are identical at the character-level while non-delimiter words match if they have identical conflation symbols at the word-level.) To be most effective as a representative, ListReader selects the member of the set of templates that are similar that has the largest "representativeness score"—ideally, the "best" representative. The representativeness score is the product of match count (the number of strings in the input text that match the field group template) and length (the number of word tokens in the field group template). Continuing, ListReader considers any remaining field group templates that have not yet been grouped together. Only field group templates that have a normalized Levenstein edit distance of at least 0.75 from every previously-chosen field group template representative will become a new set representative (meaning, the new representative overlaps perfectly with no more than 25% of the content of any existing cluster prototype).

For example, the "`[\n-Segment]`" in the first constituent position of the last cluster of records in Figure 9 has eight text-string alternatives, "`\nAgnes, 25 Sept. 1653`", "`\nJonet, 17 Sept. 1654`", ..., "`\nRose, Robert\n`". The first six of these eight field group segments have two patterns of conflated text at the word level, namely "`\n[UpLo],[Sp][DgDg][Sp][UpLo].[Sp][DgDgDgDg]`" and "`\n[UpLo],[Sp][Dg][Sp][UpLo][Sp][DgDgDgDg]`". These two patterns differ by an edit distance of less than 0.25, and thus ListReader groups these two patterns together and then chooses one of them to be the representative. ListReader would choose the first, because it has a representativeness score of $3 * 10 = 30$ whereas the other option has a representativeness score of $3 * 9 = 27$. The last two instances of the eight have the pattern "`[Sp][UpLo+],[Sp][UpLo+]`" which differs from the first chosen representative by more than an edit distance of 0.25, and thus it stands alone as a second representative of the "`[\n-Segment]`" of the third record cluster in Figure 9.

In preliminary experiments on the *Shaver-Dougherty Genealogy*, ListReader's selection procedure, as just described, improved precision, recall, F-measure, and reduced the number of required hand-labelings compared to two other policies, one that selected the single longest field group template among all alternatives and one that selected the longest field group template for each set of templates (as defined by the same two edit distance thresholds described above). This last policy performed worse than the other two. We expect this behavior will be consistent across books.

1.	[\n-Segment][in-Segment][par-of-Segment][and-Segment][\n-End-Segment]
	[\n[UpLo],[Sp][UpLo]]
	\nSteel, John
	[, in [UpLo]]
	, in Peockland
	[, par. of [UpLo]]
	, par. of Paisley
	[, and [UpLo][Sp][UpLo]]
	, and Janet Cochrane
	[, and [UpLo]]
	, and Betbiah
	[\n]
	\n
2.	[\n-Segment][par-of-Segment][and-Segment][par-m-Segment][\n-End-Segment]
	[\n[UpLo],[Sp][UpLo]]
	\nSempile, John
	[, par. of [UpLo]]
	, par. of Killellane
	[, and [UpLo][Sp][UpLo]]
	, and Isobel Morisone
	[, par. m. [DgDg][Sp][UpLo].[Sp][DgDgDgDg]]
	, par.\nm. 15 Nov. 1653
	[\n]
	\n
3.	[\n-Segment][\n-End-Segment]
	[\n [UpLo],[Sp][DgDg][Sp][UpLo].[Sp][DgDgDgDg]]
	\nAgnes, 25 Sept. 1653
	[\n [UpLo],[Sp][UpLo]]
	\nCordoner, William
	[.\n]
	.\n
	[\n]
	\n

Fig. 10. Record and Field Group Templates Derived from the Segment-level Record Clusters in Figure 9

The final record and field group templates selected from the record clusters in Figure 9 appear in Figure 10. Field group templates are grouped under the record template they belong to and are each followed by the text string (of the field group segment) used to create them. The strings also serve as example of the text they will match.

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 (token), including alphabetic words, numerals, spaces, 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 multinomial distribution—a table of conditional probabilities indicating which obser-

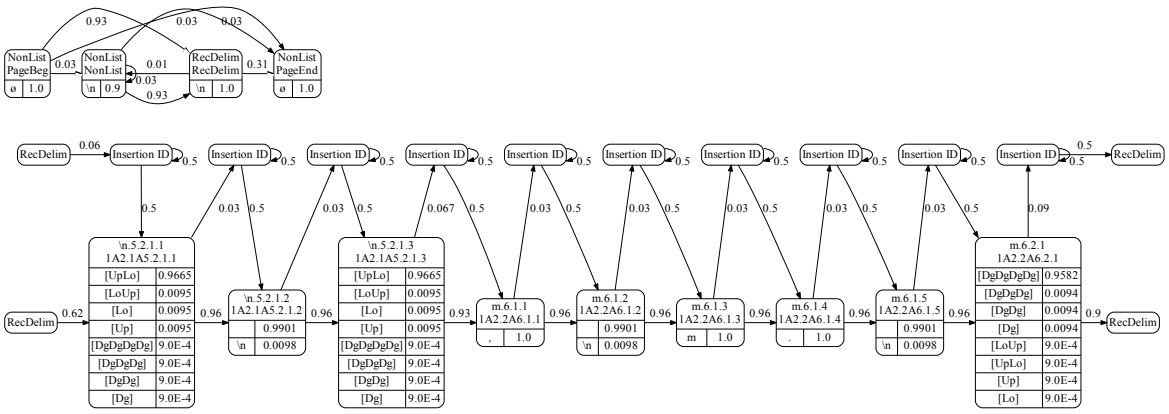


Fig. 11. HMM for text "\nIsobel Morisone, m. 1653\nJonat Allasoune, m. 1659\n".

vation w can be emitted[5] from which hidden state s and with what probability given s. The transition model is also a multinomial 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 we need to produce both the transition and emission model for our application.

Figure 11 shows an HMM constructed from the parse trees of the text "\nIsobel Morisone, m. 1653\nJonat Allasoune, m. 1659\n". Nodes represent states. The top section of each node contains the node's IDs, explained below. The bottom sections (if present) show the non-zero emission parameters for that state. Edges represent the non-zero transition parameters between states. We have duplicated the *RecDelim* node in the figure to avoid complicating the figure with long or crossing edges.

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. We now explain how ListReader automatically constructs these two levels of structure from the parsed record templates previously discovered and filtered automatically. In Subsection 3.1, ListReader generates the record-level states and their syntactic and semantic IDs. In Subsection 3.2, ListReader sets transition and emission parameters within field group templates. In Subsection 3.3 ListReader finishes the transition model connecting field groups to each other and connecting record templates and the page-level HMM states and setting the page level state's emission parameters.

[5]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.

3.1. Record-level State Generation

ListReader transforms each field group template that is selected as a representative by the process described at the end of Section 2 into a linear sequence of HMM states, one HMM state for each word token in the parse tree of the record template (i.e. the second level of the parse tree from the bottom). So, for the parse tree in Figure 7, ListReader would create HMM states for the following nodes:

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

ListReader connects adjacent HMM states with transitions, as explained later. ListReader assigns each state two IDs. These are seen as the first two lines in the nodes in Figure 11 and consist of a semantic ID and a syntactic ID. For record-level states, the semantic ID is derived from the syntactic ID, so we explain the syntactic ID first.

Each state's syntactic ID (the second identifier in the nodes of Figure 11) is a dot-delimited sequence of numbers representing the path in the parse tree from root (the record node) to word (the word for which the state is being created). Each number in this path (ignoring the number after the "A" for now) is called a parse tree number. It indicates the order of the node among its siblings in the parse tree. For example, considering the two figures Figure 7 and Figure 11, the node in the parse tree for the word `Isobel` in Figure 7 corresponds to the state "1A2.1A5.2.1.1" in the record template in the HMM in Figure 11. The syntactic ID indicates the following correspondence along a path from the record node to the word node:

[Record]	[\n-Segment]	[\n-FieldGroup]	[UpLo+]	[UpLo]
1(A2)	1(A5)	2	1	1

The record node is always assigned a parse tree number of "1" (as in "1A2") because it is always the only sibling on that level in the parse tree. The field group segment in this example is also assigned a parse tree number of "1" (as in "1A5) because it is the first child of the record node. The other parse tree numbers are assigned in similar fashion.

The first two parts of the syntactic ID are special cases since they contain another number, the alternation number (the number after the "A" in the syntactic ID). The alternation number is a number uniquely identifying the record or field group template among alternative record or field group templates discovered in Section 2. These two alternation numbers make the syntactic ID functionally complete within the HMM wrapper. The record alternation number identifies one record template among all available record templates which differ from each other in field group content or order. The combination of the record alternation number and the parse tree numbers makes each node's syntactic label unique within an HMM. For example, there is only one node with syntactic ID "1A2.1A5.2.1.1" in the HMM in Figure 11 (with or without the field group alternation number). The field group alternation number plays an important role in creating the semantic ID of a node and in improving labeling efficiency during active sampling, as we explain later.

The semantic ID of a state is the suffix of the syntactic ID starting with the field group segment's alternation number. The semantic ID, therefore, represents both a type of field group segment and a word'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 should be labeled the same by the user and therefore ListReader should request only one label from the user for all states (and matching strings within the input text) that share a semantic ID. ListReader carefully infers semantic IDs and therefore carefully assigns field group alternation numbers. ListReader assigns the

same field group alternation number to field group segments 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] [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 below, prevents the user from labeling more than one example of that field group. For example, the field group template "`, and [UpLo] [UpLo]`" matching "`, and Janet Cochrane`" in the first cluster of Figure 9 and the identical field group template "`, and [UpLo] [UpLo]`" matching "`, and Isobel Morisone`" in the second cluster 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. Record-level 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 was trained, such as word substitutions, insertions, and deletions. Therefore, beyond MLE, we also smooth these parameters using pseudo-counts (Dirichet 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 were built from text "`\nIsobel Morisone, m. 1653\n`", we still expect that HMM to match text like "`\nIsobel Morisone; m. 1653\n`", despite the semicolon replacing the comma. ListReader allows for substitutions using both conflation of text and smoothing in the emission model. The emission model of each record-level state is set with the conflated text of the word 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]`" in the seventh state on the bottom row of Figure 11. For conflated numerals and alphabetic words, their emission parameters 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, a second-to-last state in Figure 11 for a word with conflated text of "`[DgDgDgDg]`" will receive 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]`". This promotes better alignment of similar words, especially words of the same character class, despite the very small amount of training data provided to train the HMM and despite possible OCR errors and other variations. Similarly, ListReader adds pseudo-counts of spaces for the two kinds of internal space it encounters, (" " and "`\n`"), unless it's syntactic label is "record delimiter". Finally, all record-level state emission models receive a pseudo-count of 0.0001 for every other word in the document except the page beginning and ending symbols. These are omitted from the figure for simplicity.

A deletion is a sequence of one or more tokens of a record template that are missing from text that should otherwise match that record template. For example, though the HMM in Figure 11 was trained on text like "`\nIsobel Morisone, m. 1653\n`" we expect the HMM to be flexible enough to match text like "`\nIsobel Morisone m 1653\n`" (with the punctuation missing). This means that states in the HMM that were not adjacent in training data should become adjacent during execution. To allow this, 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 that pair of states satisfies our deletion constraint. The deletion constraint is an ordering on states such that the second state must follow the first state within a training record, regardless of how far apart the words were. For example, the word "Morisone" precedes the space after the comma, so the transition from the state representing "Morisone" to the state representing the space receives the non-zero pseudo-count. But the reverse transition (from " " to "Morisone") would receive a zero parameter. 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 "1A2.1A5.2.1.3" ("Morisone") and "1A2.2A6.1.1" (" ") are correctly ordered because they share a record number and their field group parse tree numbers are in the correct order. But the reverse order would not be allowed. The algorithm also checks for states in different record templates or alternate field group templates at the same position in a record template and prevents any state in one template from becoming adjacent to any state in the alternative template, regardless of the order. So, if there were an alternate initial field group in the HMM of Figure 11, its first state might have ID "1A2.1A4.2.1.1", and ListReader would prevent it from becoming adjacent to state "1A2.1A5.2.1.1" in either order. The algorithm to check if deletion order possible is the following. Split the syntactic labels into numeral pieces. Compare piece number pairs from top to bottom (left to right within label string). At the first differing piece, if they differ in alternates ("A" numbers) and this is the record piece, then return false. If they differ in the parse tree number, return true if and only if the numbers are correctly ordered and both states actually exist within the HMM. We avoid using a quadratic amount of memory by applying this order-check algorithm based on the syntax IDs instead of explicitly encoding all possible transitions in the transition model. ?? Possible transitions denoting deletions do not appear in Figure 11 because ...

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, though the HMM in Figure 11 was trained on text like "\nIsobel Morisone, m. 1653\n" we expect the HMM to be flexible enough to match text like "\nIsobel Morisone, m.. 1653.\n" (containing two extra dots). To allow insertions, ListReader must do more than alter the transition and emission models. It must introduce a new set of states, which we call insertion states. For every allowed transition between record-level states s_{t-1} and s_t (including record delimiters), ListReader creates a new state whose syntactic label is the concatenation of the two state's syntactic label $(s_t + s_{t-1})$ and whose semantic label is the concatenation of the two states' semantic labels. For example, the inserted node between "1A2.2A6.1.3" ("m") and "1A2.2A6.1.4" (".") will be "1A2.2A6.1.3 + 1A2.2A6.1.4" and its semantic ID will be "A6.1.3 + A6.1.4". ListReader sets counts for three new transitions per insertion state: one to the insertion state from the original 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 original 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 an integer between 1 and 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. Notice in the last row of states in Figure 11 that the transition between states "1A2.1A5.2.1.3" and "1A2.2A6.1.1" is smaller than the others in that row. We choose 30 because there are approximately 30 instances of a transition in normal text for every instances of an insertion at that specific location in a record template.

3.3. Connecting the Pieces

ListReader must connect field group templates to record delimiter states and to each other to complete each record template HMM. ListReader creates transitions from *RecordDelimiter* to the starting states of each of the initial field group templates with a count equal to the size of the field group template cluster (the one determined by the normalized Levenstein edit distance of 0.25). Based on the example record templates in Figure 10, ListReader would add four transitions from *RecordDelimiter* to the first state of each of the representative initial field group templates (the ones starting with "\n" in Figure 10). Then, the parameters of the transitions going from *RecordDelimiter* to each of the four initial field group segments would be initialized with the counts of the respective instances (seen in Figure 9). The count assigned to the first, second and fourth field group templates would be 2 each, and the count assigned to the third field group template would be 6. ListReader also creates a transition from the last state of each field group template to the first state of each following field group template (including *RecordDelimiter* in the case of the final field group template), with count of 1.0.

ListReader must also connect record templates to the page-level model. There are four page-level states as shown at the top of Figure 11; they always occur in any HMM ListReader constructs. They are *PageBeginning*, *PageEnding*, *NonList*, and *RecordDelimiter*. The emission model of *PageBeginning* and *PageEnding* are fixed to contain only the special character that we artificially insert 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 an allowable character is 1.0 and all other probabilities are 0.0—in other words, no parameter smoothing is allowed. The emission model of *NonList* is not fixed. Rather, it is set as the MLE estimate of all word tokens in the input text that were not covered by any candidate records during unsupervised grammar induction. We train the emission model of the non-list state on unlabeled data and the emission models of list states (above) on labeled data (specifically automatically-labeled data that we know from previous research has high precision and moderate recall). 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 there, even those not appearing in the training data. (These are not shown in Figure 11.)

The parameters for the transitions among these four page-level states are also trained using MLE from the records discovered during grammar induction. 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 have a probability of 0.1. 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 an empty page) or from *RecordDelimiter* to *PageEnding* if there were no pages ending with discovered records, for example, but not from *PageEnding* to *PageBeginning*.

4. FINAL EXTRACTION

To map data in record patterns to an ontology, ListReader does two things. First it actively and selectively requests text labels from the user by which it may associate HMM states with elements of the ontology, as explained in Subsection 4.1, and then it applies that state-label knowledge to extract information from throughout the input text and maps that information 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 who labels the fields in the text of a record matched by a part of the HMM that ListReader selects. On each iteration of the loop, the user labels the text that ListReader chooses and highlights. Actual 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 (e.g. Figure 1). ListReader then accepts the labeled text via the Web form interface and assigns labels to the corresponding HMM states, which completes 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.

This 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 learning ListReader does is fully unsupervised—no HMM structure or parameter learning takes place under the supervision of a user either interactively or in advance. 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 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 sub-HMM with the highest predicted return on investment (ROI). A selected sub-HMM must be part of a record template. When there is more than one string that matches the best sub-HMM, ListReader selects the first one on whichever page contains the most matches of that sub-HMM. One can think of ROI as the slope of the learning curve: higher accuracy and lower cost produce higher ROI. ListReader computes predicted ROI as the sum of the counts of the strings matching each state in the candidate sub-HMM divided by the number of states in the sub-HMM. It limits the set of candidate sub-HMMs to those that are contiguous and complete, meaning sub-HMMs that contain no record delimiters or states already labeled by the user and that are not contained by any longer candidate sub-HMM. This is so ListReader queries the user once instead of multiple times for a section of text, a part of which might have a larger predicted ROI than the whole string. Querying the user to maximizing the immediate ROI tends to maximize the slope of the learning curve and has proven effective in other active learning situations [Haertel et al. 2008]. Once labeling of the selected text is complete, ListReader removes the counts for all strings that match the corresponding capture groups, recomputes the ROI scores of remaining capture groups, and issues a query to the user.

As an example, supposed ListReader's HMM discovers the same groups of records as in Figure 9. If a majority of the shorter, child records appear on the same page

(as they do), ListReader would select the first one, "`Agnes, 25 Sept. 1653.`" in the first cycle of active sampling, because the predicted ROI of labeling this text is $(10 * 3 + 9 * 3)/10 = 5.7$, while the predicted ROI of labeling the best alternative ("`Steel, John, in Peockland, par. of Paisley, and Betbiah`") is $(22 * 2)/22 = 2$.

When one HMM state receives a user-supplied label, all states sharing the same semantic ID receive the same final label. This minimizes the labeling effort during active sampling. Therefore, the counts on which the query policy is based are aggregates of matches for all states sharing a semantic ID.

Active sampling is impactful from the very first query and improves recall nearly 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 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 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 labeled text strings into predicates that it inserts into the ontology (excluding tokens labeled as "NonList". The entire flow from HTML form and text to ontology 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). This effectively reduces 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 resulting ontology. The leaves of the tree of form fields are lexical text-entry fields into which the user inserts field text from the page. 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 $n > 1$) between the corresponding object sets. The root of the tree represents the primary object set, i.e. the topic of a record in a list, for example a person in Figure 1.

ListReader maps the empty HTML form to an ontology schema that may contain a number of expressive constructs including any of the following. (1) textual vs. abstract entities (e.g. *GivenName*("John") 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*s 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; (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); and (5) concept categorization hierarchies, including, in particular, role designations. This expressiveness provides for the rich kinds of fact assertions we wish to extract in our application.

When the user fills in the HMTL form during each iteration of active sampling, ListReader maps the text of the whole page plus the filled-in form to an in-line labeled text format for the page. Figure 3 shows an example of just one labeled portion of a page. Labels indicate a path from the root (primary object of a record) to a leaf node (a lexical field of that record) by naming all the object sets (form field names) in the path using a dot-separated list of object set names. Additional notation distinguishes between predicates of different arity and cardinality and will include the names of additional object sets that participate in the relationship in parentheses (since a single path in a tree would normally name only the object sets in binary relationships).

When receiving labeled text from the form UI, ListReader first tokenizes the in-line labeled text of the page, attaches the tag labels to each token, and associates each label with a corresponding HMM syntactic ID (the one that matched the token during the first application of the HMM). During the second (also the last) application of the HMM after active sampling is complete, ListReader produces the same kind of in-line labeled text of a page from the completed HMM wrapper applied to the whole page and translates the labeled text into predicates. This is done by splitting the token labels into object set names and instantiating objects for each new object set name and relationship predicates for each dot-separated set of object set names. The text string of the each field is instantiated as a lexical object. Object sets with the same name are instantiated as the same object within a given record unless the object set name is subscripted, with the effect that all objects instantiated for a record are tied by relationships back to the same primary object. Record delimiter tags that surround a complete record string determine which fields belong to the same record. Any remaining unlabeled text (text labeled as "NonList") produces no output.

5. EVALUATION

This section evaluates ListReader on two books, the *Shaver-Dougherty Genealogy* and the *Kilbarchan Parish Register*, and compares 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 HMMs. The previous version of ListReader is similar to the current version in other respects, except that it created regex wrappers for every pattern discovered during grammar induction whereas the newer version is selective in which record and field group templates make it into the final HMM wrapper, and the regex wrappers are brittle whereas the HMM wrappers are not so that regex wrappers only match text identically whereas HMM wrappers match text with a degree of variation allowed.

Below, we describe the data (books) we used to evaluate ListReader in Subsection 5.1. We explain the experimental procedure comparing ListReader's performance with the performance of 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.[6]

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*.

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 shown 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 all 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. 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 12 shows the ontology corresponding to those 46 field labels. 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.

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 much greater amount of knowledge engineering than we allow for ListReader. We simulated active learning of a CRF using a random sampling strategy—considered

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

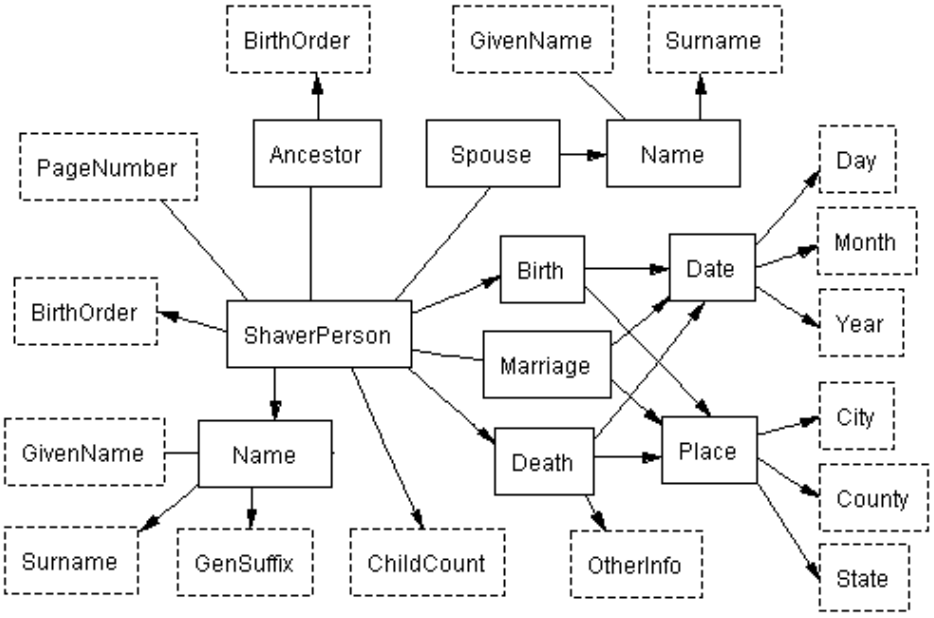


Fig. 12. ShaverPerson Ontology. *ShaverPerson* is the primary object set.

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 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 but ListReader never does. To be consistent in measuring cost,

we do not count these labelings against the cost for the CRF. This means that the CRF has a slight advantage as it receives training data for negative examples (prose text) without affecting its measured cost. During the test phase, the evaluation system measured the accuracy of the extractors only on tokens of text not labeled for training or active sampling.

Since our aim is to develop a system that accurately extracts information at a low cost to the user, our evaluation centers on a standard metric in active learning research that combines both accuracy and cost into a single measurement: Area under the Learning Curve (ALC) [Cawley 2011]. The rationale is that there 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 (p) is defined to be $\frac{tp}{tp+fp}$ and recall (r) 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 and recall, or $\frac{2pr}{p+r}$. ALC is $\int_{min}^{max} f(c)dc$, wehre c is the number of user-labeled fields (cost) and $f(c)$ can be precision, recall, or F-measure as a function of cost. *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, 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,[7] 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 produced very few false positives (precision errors). The improvement of ListReader (HMM) over (Regex) is due to improved recall. The initial HMM is capable of recognizing up to 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 that satisfy our record selection constriants while the Regex preserves all of them. ListReader (HMM) does not produce as high a precision, 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 13, 14, and 15 show plots of the F-measure, precision, and recall learning curves for ListReader and the CRF on the *Shaver-Dougherty Genealogy* and Figures 16, 17, and 18 show plots of the F-measure, precision, and recall learning curves for ListReader and the CRF on the *Kilbarchan Parish Register*. These plots provide

[7] 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.

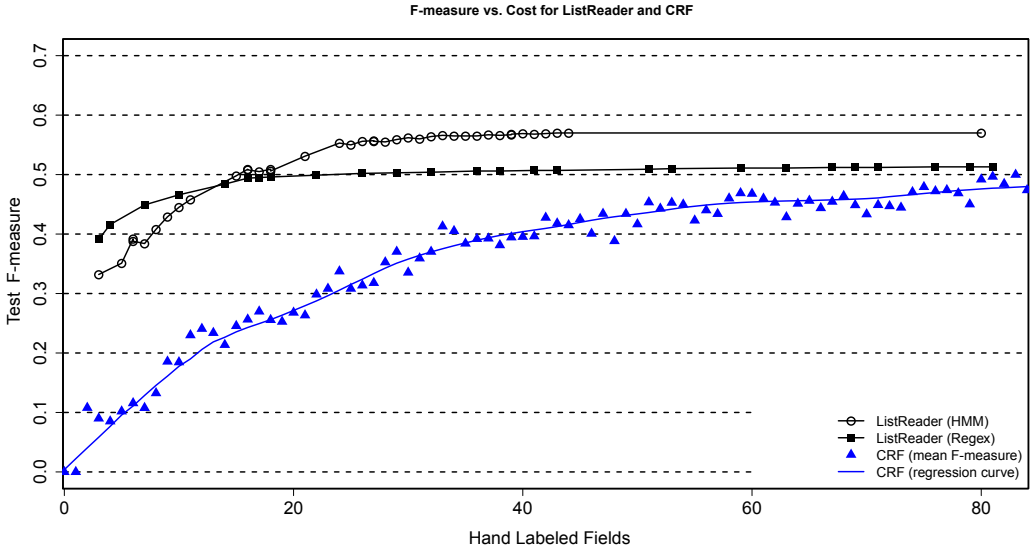


Fig. 13. F-measure Learning Curves for the *Shaver-Dougherty Genealogy*

detail behind the ALC metrics in Tables I and II. Visually, the comparative learning curves indicate that ListReader (Regex or HMM) outperforms the CRF fairly consistently over varying numbers of field labels for all three metrics. Tables I and II tells us that the differences among the three extractors are statistically significant for most pairwise comparisons at $p < 0.05$ using an unpaired t test. The three pairs that are not significant are the ones comparing the recall of ListReader (Regex) and the CRF on both the *Shaver-Dougherty Genealogy* and the *Kilbarchan Parish Register* and comparing the precision of the two versions of ListReader on the *Kilbarchan Parish Register*.

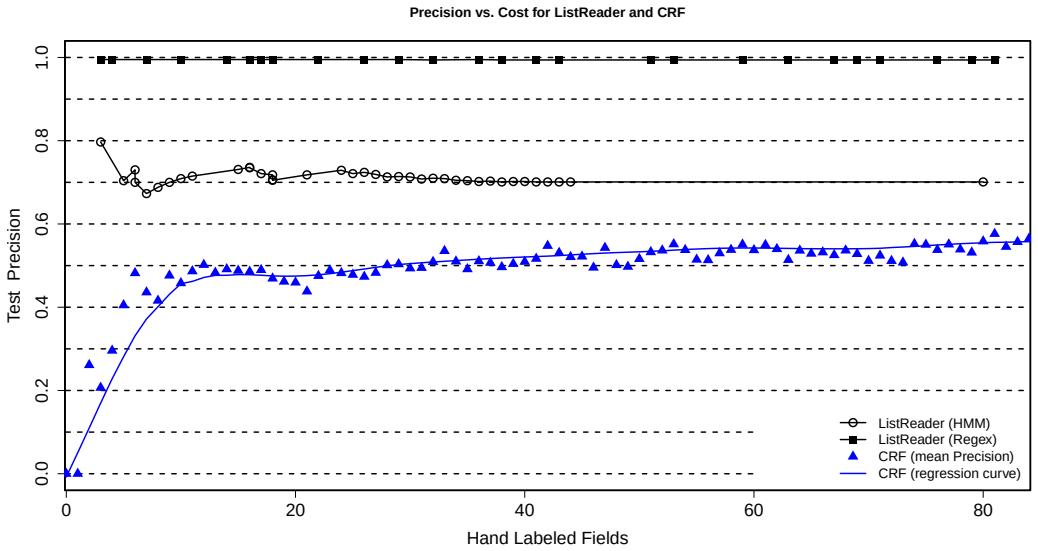


Fig. 14. Precision Learning Curves for the *Shaver-Dougherty Genealogy*

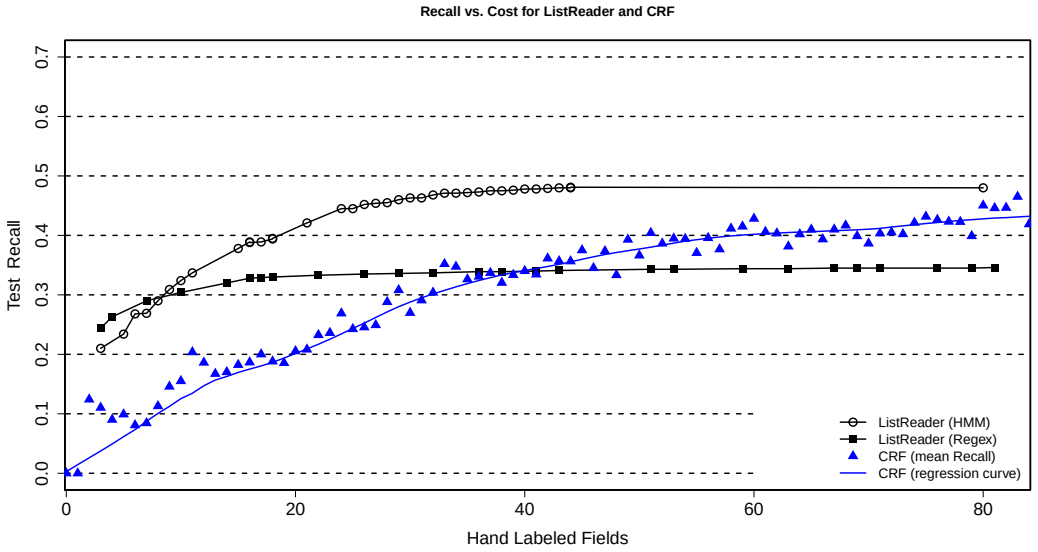


Fig. 15. Recall Learning Curves for the *Shaver-Dougherty Genealogy*

The spike in the CRF's recall at Cost = 4 in Figure 18 is because the majority of records in the book are child records which contain 4 fields. When the CRF is lucky enough to train on one of these records, it usually does well extracting all the other child records.

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

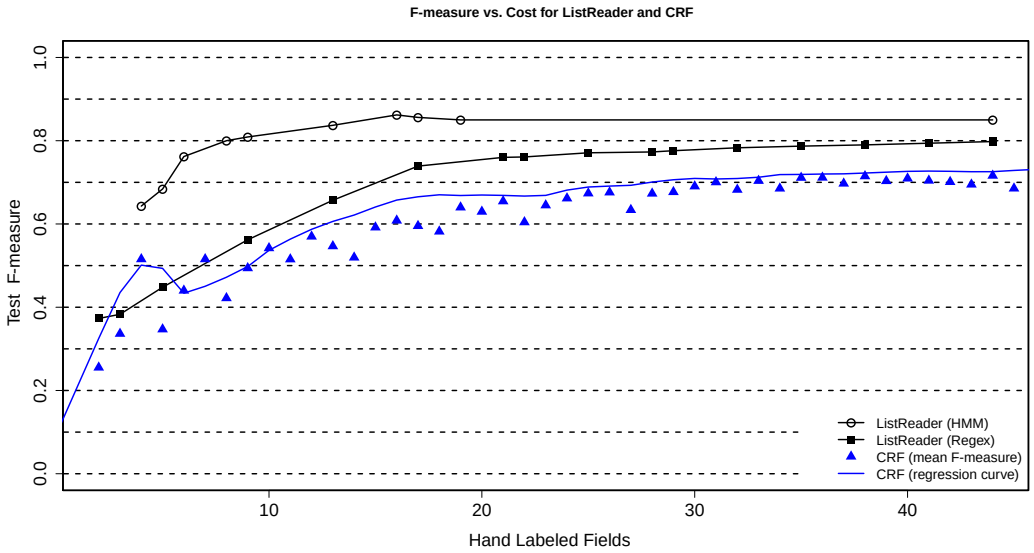


Fig. 16. F-measure Learning Curves for the *Kilbarchan Parish Register*

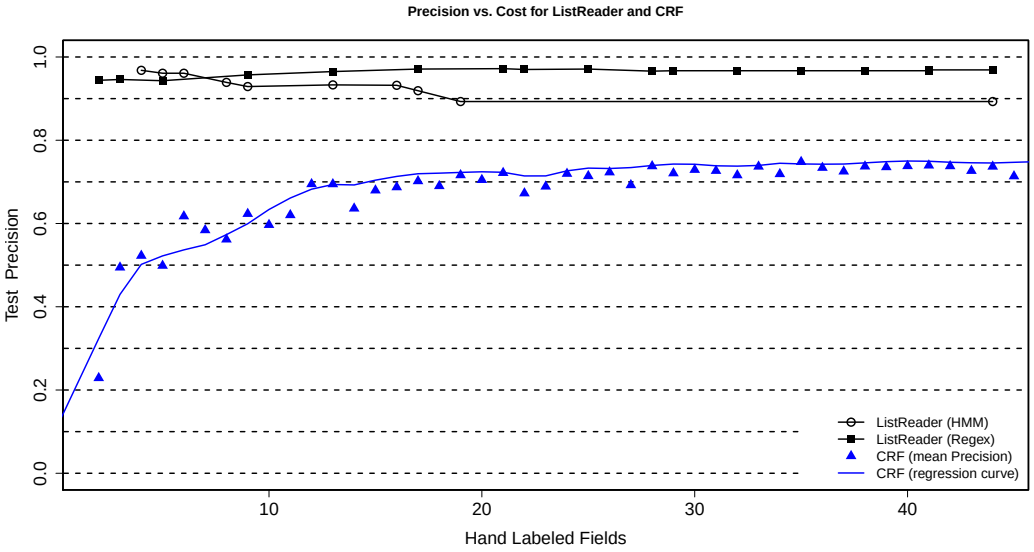


Fig. 17. Precision Learning Curves for the *Kilbarchan Parish Register*

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 HMM record

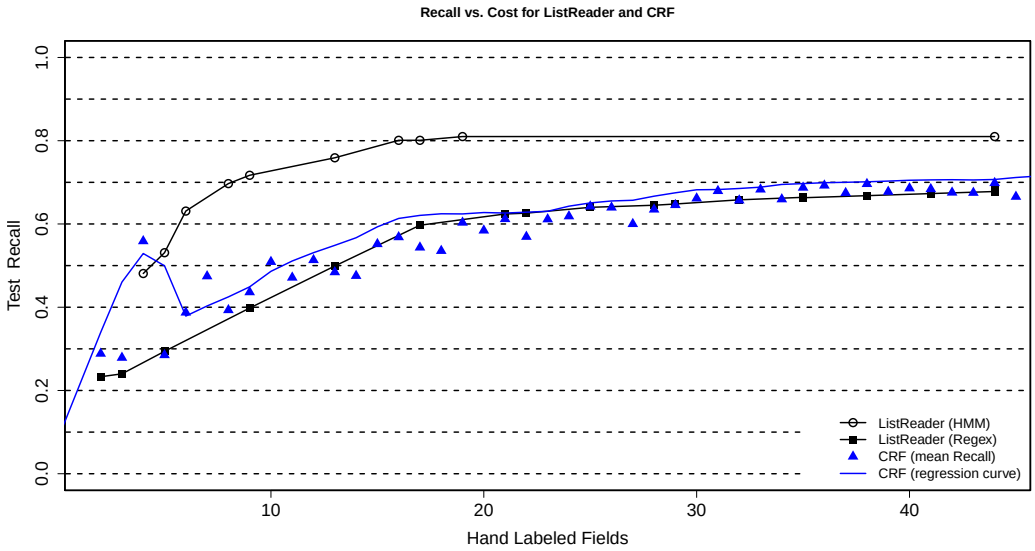


Fig. 18. Recall Learning Curves for the *Kilbarchan Parish Register*

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

Comparing the running time of the extractors, 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 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*. CRF took 52 seconds on *Shaver-Dougherty Genealogy* and 9 seconds on *Kilbarchan Parish Register*.

6. DISCUSSION AND FUTURE WORK

In our error analysis we see that ListReader (HMM) produced 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 marker 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 just the right information". The technique of using semantic or lexical resources is complicated in our work by the common OCR errors that make dictionary matching more difficult.

On the other hand, adding semantic constraints to the HMM would likely help prevent some of its precision errors, such as labeling an "m." as a surname at the beginning of a line that otherwise matched a known record pattern. Future work should investigate adding such semantic features or constraints to ListReader in a way that is cost-effective, for example using self-supervision, e.g. a bootstrapping mechanism that learns its own semantic categories by combining multiple sources of evidence in an expanded set of input text. 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 very scalable in terms of supervision cost.

The HMM is currently limited in how it utilizes user feedback. For example, the generic insertion states may match many different textual patterns that will not necessarily have the same final labels. Future work should change ListReader so it creates a sub-HMM for each pattern matched by a given insertion state.

7. RELATED WORK

We now compare our current work with other research having a strong component of unsupervised learning in the context of information extraction applied to lists. These works are almost universally applied to clean text, and most of them are applied to structured HMTL documents and would therefore not perform well on noisy OCR text that lacks HTML structure.

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 unsupervised or semi-supervised learning scenarios. In either case, text without manually-provided labels are assigned those labels that are most probable given the current HMM parameters, and those HMM parameters are in turn set from the most probable label distributions given the parameters set on the previous iteration. Grenager

et al. ([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 genre-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.[8] 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.

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 in some the recent web wrapper papers 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 numbers of fields 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

[8]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.

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.

ListReader solves 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.

8. CONCLUSIONS

ListReader has demonstrated 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 has also demonstrated 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 F-measure as a function of labeling cost with statistical significance.

APPENDIX A: Example Pages

We now show an example page from *The Ely Ancestry* in Figure 19, from the *Shaver-Dougherty Genealogy* in Figure 20, and three pages from the *Kilbarchan Parish Register* in Figures 21, 22, and 23.

ACKNOWLEDGMENTS

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

REFERENCES

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

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

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

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.

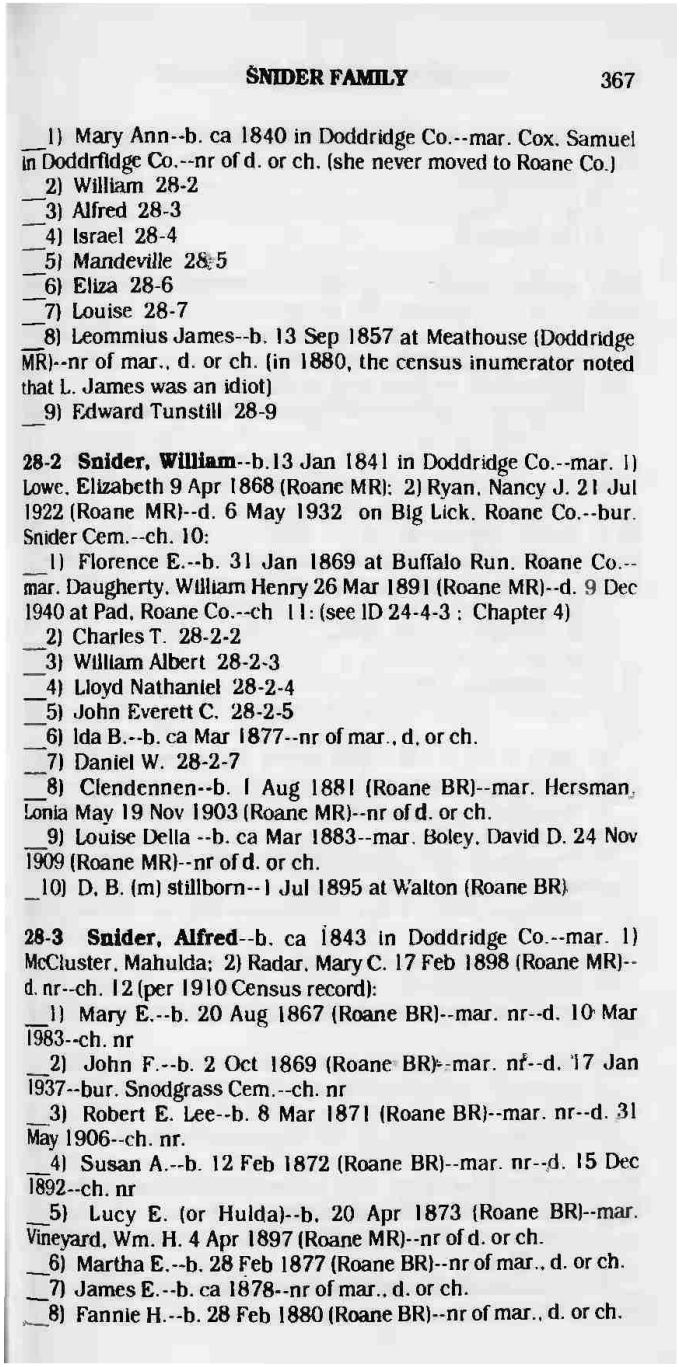
The attempt was gallantly made, but was only partially successful, owing to the ship's thick copper sheathing.

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. 19. Page from *The Ely Ancestry*, Page 367



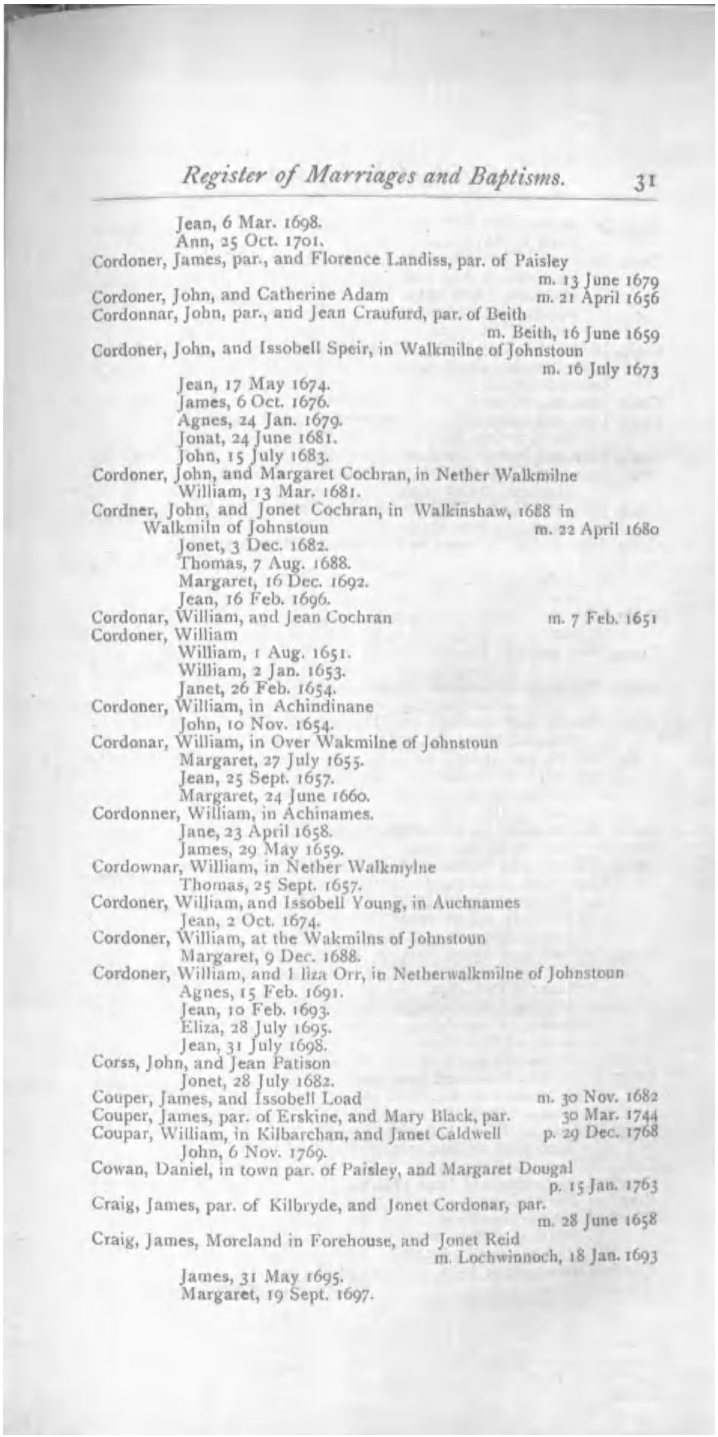
SNIDER FAMILY 367

__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.

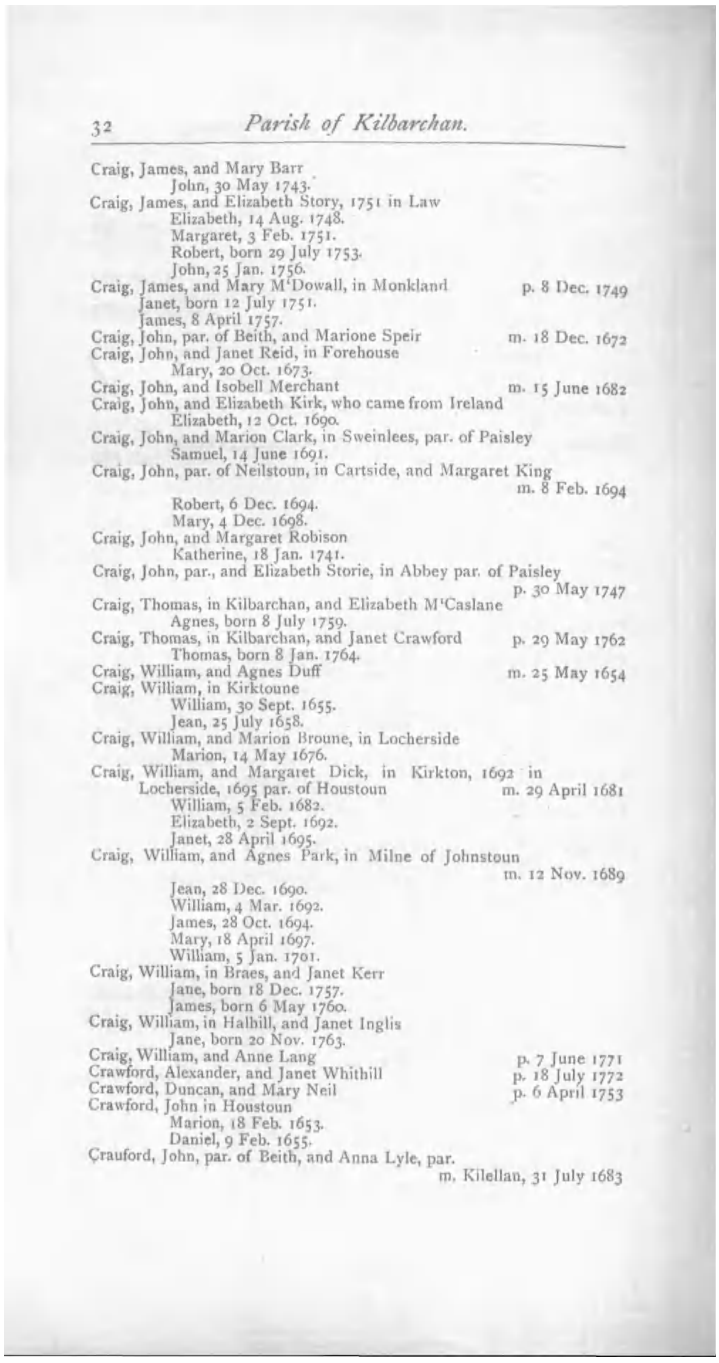
Fig. 20. Page from *Shaver-Dougherty Genealogy*, Page 154



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.

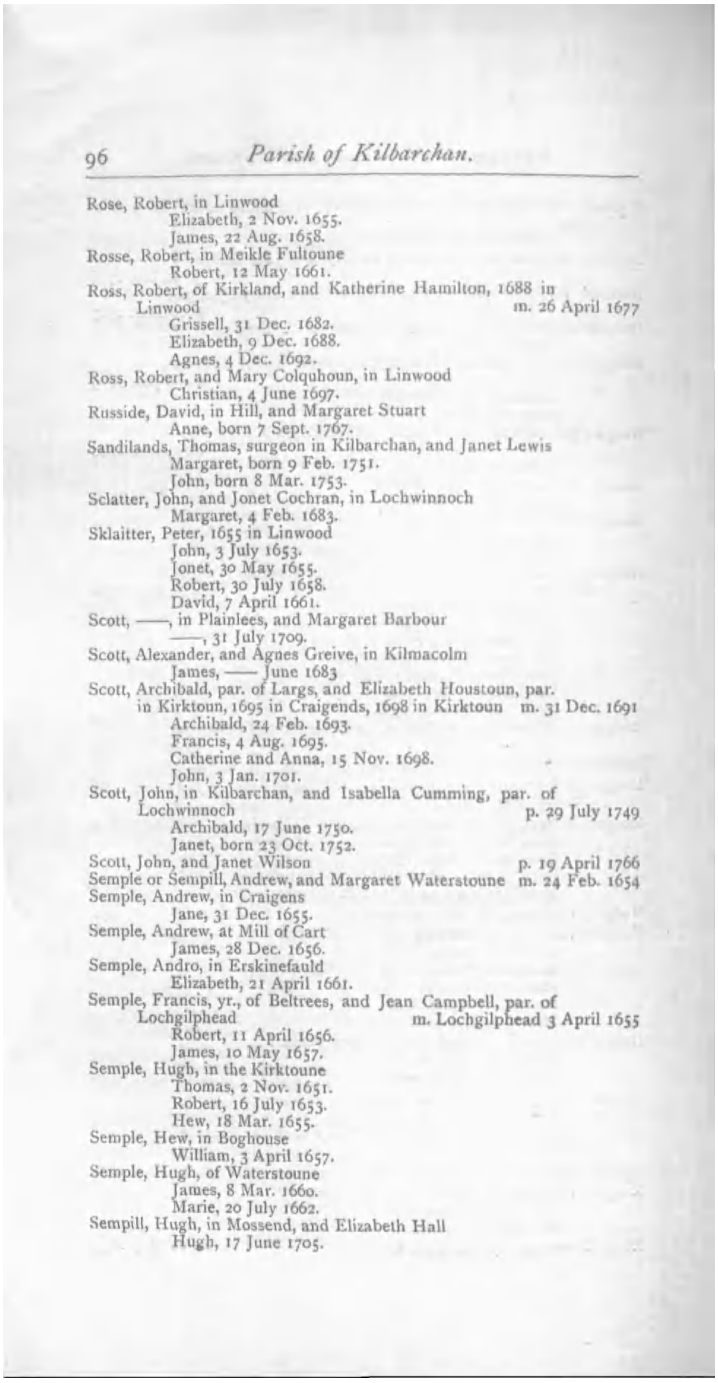
Fig. 21. Page from *Kilbarchan Parish Register*, Page 31



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. 22. Page from *Kilbarchan Parish Register*, Page 32



96 *Parish of Kilbarchan.*

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

Fig. 23. Page from *Kilbarchan Parish Register*, Page 96

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 AF Park. 2008. Error Correcting Output Coding-Based Conditional Random Fields for Web Page Prediction. In *Web Intelligence and Intelligent Agent Technology, 2008. WI-IAT'08. IEEE/WIC/ACM International Conference on*, Vol. 1. IEEE, 743–746. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4740540

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

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

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

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.

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

Y. Li, J. Jiang, H.L. Chieu, and K.M.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/

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.

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

Received Month 0000; revised Month 0000; accepted Month 0000