Using Language Models for Flat Text Queries in XML Retrieval

Paul Ogilvie, Jamie Callan
Language Technologies Institute
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA USA
{pto,callan}@cs.cmu.edu

ABSTRACT

This paper presents a language modeling system for ranking flat text queries against a collection of structured documents. The retrieval system, built using the Lemur toolkit, produces probability estimates that arbitrary document components generated the query. This paper describes storage mechanisms and retrieval algorithms for the evaluation of unstructured queries over XML documents. The paper includes retrieval experiments using a generative language model on the content only topics of the INEX testbed, demonstrating the strengths and flexibility of language modeling to a variety of problems. We also describe index characteristics, running times, and the effectiveness of the retrieval algorithm.

1. INTRODUCTION

Language modeling has been studied extensively in standard Information Retrieval in the last few years. Researches have demonstrated that the framework provided by language models has been powerful and flexible enough to provide strong solutions to numerous problems, including ad-hoc information retrieval, known-item finding on the Internet, filtering, distributed information retrieval, and clustering.

With the success of language modeling for this wide variety of tasks and the increasing interest in studying structured document retrieval, it is natural to apply the language modeling framework to XML retrieval. This paper describes experiments using one way the generative language model could be extended to model and support queries on structured documents. We model documents using a tree-based language model. This is similar to many previous models for structured document retrieval [1][2][3][6][7][10], but differs in that language modeling provides some guidance in combining information from nodes in the tree and estimating term weights. This work is also similar to other works using language models for XML retrieval [5][9], but differs in that we also present context-sensitive language model smoothing and an implementation using information retrieval style inverted lists rather than a database.

The next section provides background in language modeling in information retrieval. In Section 3 we present our approach to modeling structured documents. Section 4 describes querying the tree-based language models presented in the previous section. In Section 5, we describe the indexes required to support retrieval and the retrieval algorithms. We describe the experiment setup and indexes used for INEX 2003 in Section 6. Section 7 describes experimental results. We discuss relationships to other approaches to structured document retrieval in Section 8, and Section 9 concludes the paper.

2. LANGUAGE MODELS FOR DOCUMENT RETRIEVAL

Language modeling applied to information retrieval problems typically models text using unigram language models. Unigram language models are similar to bags-of-words representations, as word order is ignored. The unigram language model specifically estimates the probability of a word given some text. Document ranking typically is done one of two ways: by measuring how much a query language model diverges from document language models [8], or by estimating the probability that each document generated the query string. Since we use the generative language model for our experiments, we will not describe the divergence based approaches here.

2.1 The Generative Language Model

The generative method ranks documents by directly estimating the probability of the query using the texts' language models [13][4][15][16]:

$$P(Q|\theta_T) = \prod_{w \in Q} P(w|\theta_T)^{qtf(w)}$$

where Q is the query string, and θ_T is the language model estimated for the text, and *qtf*(*w*) is the query term frequency of the term *w* (count of *w* in the query). Texts more likely to have produced the query are ranked higher. It is common to rank by the log of the generative probability as it there is less danger of underflow and it produces the same orderings:

$$\log(P(Q|\theta_T)) = \sum_{w \in Q} qtf(w) \log P(w|\theta_T)$$

Under the assumptions that query terms are generated independently and that the query language model used in KL-divergence is the maximum-likelihood estimate, the generative model and KL divergence produce the same rankings [11].

2.2 The Maximum-Likelihood Estimate of a Language Model

The most direct way to estimate a language model given some observed text is to use the maximum-likelihood estimate, assuming an underlying multinomial model. In this case, the maximum-likelihood estimate is also the empirical distribution. An advantage of this estimate is that it is easy to compute. It is very good at estimating the probability distribution for the language model when the size of the observed text is very large. It is given by:

$$P_{MLE}(w|\theta_T) = \frac{freq(w, T)}{|T|}$$

where T is the observed text, *freq*(*w*, T) is the number of times the word *w* occurs in T, and |T| is the length in words of T. The maximum likelihood estimate is not good at estimating low frequency terms for short texts, as it will assign zero probability to those words. This creates a problem for estimating document language models in both KL divergence and generative language model approaches to ranking documents, as the log of zero is negative infinity. The solution to this problem is smoothing.

2.3 Smoothing

Smoothing is the re-estimation of the probabilities in a language model. Smoothing is motivated by the fact that many of the language models we estimate are based on a small sample of the "true" probability distribution. Smoothing improves the estimates by leveraging known patterns of word usage in language and other language models based on larger samples. In information retrieval smoothing is very important [16], because the language models tend to be constructed from very small amounts of text. How we estimate low probability words can have large effects on the document scores. In addition to the problem of zero probabilities mentioned for maximum-likelihood estimates, much care is required if this probability is close to zero. Small changes in the probability will have large effects on the logarithm of the probability, in turn having large effects on the document scores. Smoothing also has an effect similar to inverse document frequency [4], which is used by many retrieval algorithms.

The smoothing technique most commonly used is linear interpolation. Linear interpolation is a simple approach to combining estimates from different language models:

$$\mathrm{P}\left(w|\theta\right) = \sum_{i=1}^{k} \lambda_i \mathrm{P}\left(w|\theta_i\right)$$

where k is the number of language models we are combining, and λ_i is the weight on the model θ_i. To ensure that this is a valid probability distribution, we must place these constraints on the lambdas:

$$\sum_{i=1}^{k} \lambda_i = 1 \quad \text{and for } 1 \le i \le k,\ \lambda_i \ge 0$$

One use of linear interpolation is to smooth a document's language model with a collection language model. This new model would then be used as the smoothed document language model in either the generative or KL-divergence ranking approach.

2.4 Another Characterization

When we take a simple linear interpolation of the maximum likelihood model estimated from text and a collection model, we can also characterize the probability estimates as:

$$\mathrm{P}\left(w|\theta_{\mathrm{T}}\right) = \begin{cases} \mathrm{P}_{\mathrm{seen}}\left(w|\theta_{\mathrm{T}}\right) & \textit{if } w \in \mathrm{T} \\ \mathrm{P}_{\mathrm{unseen}}\left(w|\theta_{\mathrm{T}}\right) & \textit{otherwise} \end{cases}$$

where

$$\mathrm{P}_{\mathrm{seen}}\left(w|\theta_{\mathrm{T}}\right) = (1-\omega)\mathrm{P}_{\mathrm{MLE}}\left(w|\theta_{\mathrm{T}}\right) + \omega \mathrm{P}\left(w|\theta_{collection}\right)$$

and

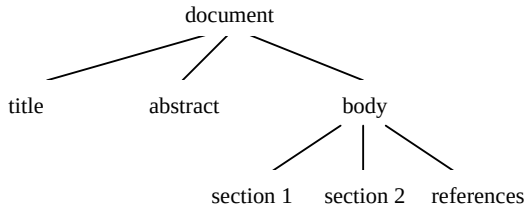
$$\mathrm{P}_{\mathrm{unseen}}\left(w|\theta_{\mathrm{T}}\right) = \omega \mathrm{P}\left(w|\theta_{collection}\right)$$

This notation distinguishes the probability estimates for cases where the word has been seen in the text and where the word has not been seen will be in the sample text. We will use this notation later when describing the retrieval algorithm, as it simplifies the description and is similar to the notation used in previous literature [16]. The simple form of linear interpolation where ω is a fixed constant is often referred to as Jelinek-Mercer smoothing.

3. STRUCTURED DOCUMENTS AND LANGUAGE MODELS

The previous section described how language modeling is used in unstructured document retrieval. With structured documents such as XML or HTML, we believe that the information contained in the structure of the document can be used to improve document retrieval. In order to leverage this information, we need to model document structure in the language models.

We model structured documents as trees. The nodes in the tree correspond directly with tags present in the document. A partial tree for a document might look like:



Nodes in the document tree correspond directly to XML tags in the document. For each document node in the tree, we estimate a language model. The language models for leaf nodes with no children can be estimated from the text of the node. The language models for other nodes are estimated by taking a linear interpolation of a language model formed from the text in the node (but not in any of its children) and the language models formed from the children.

We have not specified how the linear interpolation parameters for combining language models in the document tree should be chosen. This could be task specific, and training may be required. The approach we will adopt in this paper is to set the weight on a child node as the accumulated length of the text in the child divided by the accumulated length of the node. By accumulated length we mean the number of words directly in the node plus the accumulated length of the node's children. Setting the parameters in this manner assumes that a word in a one node type is no more important than a word in any other node type; it is the accumulated length of the text in the node that determines how much information is contained in the node.

We also wish to smooth the maximum likelihood models that are estimated directly from the text with a collection language model. In this work, we will combine the maximum likelihood models with the collection model using a linear interpolation with fixed weights. The collection model may be specific to the node type, giving context sensitive smoothing, or the collection model may be one large model estimated from everything in the corpus, giving a larger sample size.

When the λ parameters are set proportional to the text length and a single collection model is used, this results a special case that is very similar to the models used in [5][9]. The tree-based language model estimated using these parameter settings will

be identical to a language model estimated by taking a simple linear interpolation of a maximum likelihood estimate from the text in the node and its ancestors and the collection model.

4. RANKING THE TREE MODELS

In a retrieval environment for structured documents, it is desirable to provide support for both structured queries and unstructured, free-text queries. It is easier to adapt the generative language model to structured documents, so we only consider that model in this paper. It is simpler to support unstructured queries, so we will describe retrieval for them first.

4.1 Unstructured Queries

To rank document components for unstructured queries, we use the generative language modeling approach for IR described in Section 2. For full document retrieval, we need only compute the probability that the document language model generated the query. If we wish to return arbitrary document components, we need to compute the probability that each component generated the query.

Allowing the system to return arbitrary document components may result in the system stuffing the results list with many components from a single document. This behavior is undesirable, so a filter on the results is necessary.

One filter we employ takes a greedy approach to preventing overlap among components in the results list. For each result, it will be thrown out of the results if there is any component higher in the ranking that is an ancestor or descendent of the document component under consideration.

4.2 Structured Queries

Our previous paper on this subject [11] discusses how some structural query operators could be included in the model. We do not currently support any of these operators in our system, so we will not discuss in depth here. However, we will mention that the retrieval framework can support most desired structural query operators using relatively easy to implement query nodes.

4.3 Prior Probabilities

Given relevance assessments from past topics, we can estimate prior probabilities of the document component being relevant given its type. Another example prior may depend on the length of the text in the node. A way to incorporate this information is to rank by the probability of the document node given the query. Using Bayes rule, this would allow us incorporate the priors on the nodes. The prior for only the node being ranked would be used, and the system would multiply the probability that the node generated the query by the prior:

$$P(N|Q) = P(Q|\theta_N)\,P(N)/P(Q)$$

which is proportional to

$$P(Q|\theta_N)\,P(N)$$

This would result in ranking by the probability of the document component node given the query, rather than the other way around.

5. STORAGE AND ALGORITHMS

This section describes how we support structured retrieval in the Lemur toolkit. We first describe the indexes built to support retrieval. Then we describe how the indices are used by the retrieval algorithm. We also present formulas for the computation of the generative probabilities we estimate for retrieval.

5.1 Index Support

There are two main storage structures in Lemur that provide the support necessary for the retrieval algorithm. Lemur stores inverted indexes containing document and node occurrences and document structures information.

5.1.1 Inverted Indexes

The basic idea to storing structured documents in Lemur for retrieval is to use a modified inverted list. Similar to storing term locations for a document entry in an inverted list, we store the nodes and the term frequencies of the term in the nodes in the document entries of the inverted list. The current implementation of the structured document index does not store term locations, but could be adapted to store term locations in the future.

The inverted lists are keyed by term, and each list contains the following:

- document frequency of the term
- a list of document entries, each entry containing
  - document id
  - term frequency (count of term in document)
  - number of nodes the term occurs in
  - a list of node entries, each entry containing
    - node id
    - term frequency (count of term in node)

When read into memory, the inverted lists are stored in an array of integers. The lists are stored on disk using restricted-variable length compression and delta-encoding is applied to document ids and node ids. In the document entry lists, the documents entries are stored in order by ascending document id. The node entry lists are similarly stored in order by increasing node id. Document entries and node entries are only stored in the list when the term frequency is greater than zero. Access to the lists on disks is facilitated with an in-memory lookup table for vocabulary terms.

There is also an analogous set of inverted lists for attribute name/value pairs associated with tags. For example, if the document contained the text

<date calendar="Gregorian">,

the index would have an inverted list keyed by the triple date/calendar/Gregorian. The structure and information stored in the inverted lists for the attribute name/value pairs is identical to those in the inverted lists for terms.

5.1.2 Document Structure

The document structure is stored compressed in memory using restricted variable length compression. A lookup table keyed by document id provides quick access to the block of compressed memory for a document. We choose to store the document structure in memory because it will be requested often during retrieval. For each document, a list of information about the document nodes is stored. For each node, we store:

- parent of the node
- type of node
- length of the node (number of words)

Since this list of information about the document structure is compressed using a variable length encoding, we must

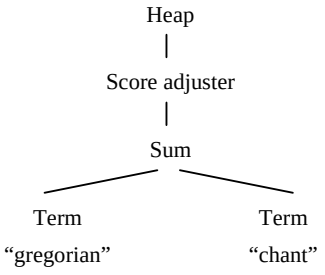
decompress the memory to provide efficient access to information about nodes. When the document structure for a document is being decompressed, we also compute:

- accumulated length of the node (length of text directly in the node + accumulated length of children)
- number of children of the node
- a list of the node's children

This decompression and computation of other useful information about the document structure is computed in time linear to the number of nodes in the document being decompressed.

5.2 Retrieval

We construct a query tree to process and rank document components. A typical query tree is illustrated below. The leaf nodes of the query tree are term nodes which read the inverted lists for a term off of disk and create result objects for document components containing the term. The term nodes are also responsible for propagating the term scores up the document tree. The sum node merges the result lists returned by each of the term nodes, combining the score estimates. The score adjuster node adjusts the score estimates to get the generation probabilities and also applies any priors. The heap node maintains a list of the top *n* ranked objects and returns a sorted result list. Efficient retrieval is achieved using a document at a time approach. This requires that the query tree be walked many times during the evaluation of a query, but results a large saving of memory, as only the result objects for a document and the top *n* results objects in the heap must be stored at any point in time.



A more detailed description of each of the query nodes follows. When each query node is called, they are passed a document id to evaluate. In order to know which document should be processed next, the term nodes pass up the next document id in the inverted list. For other query nodes, the minimum next document id among a node's children gets passed up the query tree with the results list. We will describe the query nodes bottom up, as that is how the scores are computed.

We first note that we can rewrite the log of the probability that the document node generated the query as

$$\log\left(P\left(Q|\theta_{node}\right)\right) = \sum_{w \in Q, node} qtf(w) \log\left(\frac{P_{seen}\left(w|\theta_{node}\right)}{P_{unseen}\left(w|\theta_{node}\right)}\right) + \sum_{w \in Q} qtf(w) \log P_{unseen}\left(w|\theta_{node}\right)$$

as shown in [16]. This will allow us to easily compute the item in the first sum easily using term nodes, combine these components of the score using a sum node, and then add on the rest using a score adjustment node.

5.2.1 Term Node

The term nodes read in the inverted lists for a term *w* off of disk and create a list of results where the score for a result is initialized to

$$qtf(w) \cdot \log\left(\frac{P_{seen}\left(w|\theta_{node}\right)}{P_{unseen}\left(w|\theta_{node}\right)}\right)$$

The term node assumes that the parent id of a node is smaller than the node's id. It also assumes that the document entries in inverted lists are organized in increasing document id order and the node entries are organized in increasing term id order. The structured document index we built is organized this way. In the following algorithm description, indentation is used to denote the body of a loop.

1 Seek to the next entry in the inverted list where the document id is at least as large as the requested document

2 If the document id of the next entry is the requested document

3 Decompress the document structure information for the document

4 Read in the node entries from the inverted list

5 Create the result objects for the leaf nodes. For each node that contains the term:

6 Initialize the score for the result to the seen probability part for the node

$$seen(node) = (1-\omega)\, freq(w, node)\, \lambda(node, node)$$

where

$$\lambda(node, node) = \frac{length(node)}{accumulated\ length(node)}$$

and ω will be used to set the influence of the collection models.

7 Push the node id onto the candidate node heap

8 Store the result object in an array indexed by node id for fast access

9 While the candidate node heap isn't empty:

10 Pop the top node id off of the heap (the largest node id), set it to the current node id

11 Lookup the result from the result array

12 Lookup the node id for the parent of the current node

13 Lookup the parent node's result

14 If the parent node's result object is NULL:

15 Create a new result object for the parent node and put it in the result array, initializing the score to 0

16 Push the parent node's id onto the candidate node heap

17 Propagate the seen part of the score from the current node to the parent node, setting the parent node's seen part to

$$seen(parent) + seen(node)\lambda(node, parent)$$

where

$$\lambda(node, parent) = \frac{accumulated\ length(node)}{accumulated\ length(parent)}$$

18 Push the result onto the front of the results list

19 Set the result in the result array for the node to NULL (initializing the result array for the next document)

[*Now each document node that contains the query term (or has a child containing the term) has a result in the results list where the score is the seen probability part for the query term*]

20 For each node in the result list

21 Compute the unseen part of the generative probability for each node. For linear interpolation with a constant ω and one single node type independent collection model, this is

$$unseen(w, node) = \omega \mathrm{P}(w|\theta_{collection})$$

For linear interpolation with a constant ω and node type specific collection models, this can be computed recursively

$$unseen(w, node) = \omega \mathrm{P}(w|\theta_{collection, type(node)})\lambda(node, node) + \sum_{child \in children(node)} unseen(w, child)\lambda(child, node)$$

22 Set the score for the result to

$$qtf(w) \cdot \log\left(\frac{seen(node) + unseen(w, node)}{unseen(w, node)}\right)$$

23 Return the result list and the next document id in the inverted list

The result list now contains results for a single document where the score is

$$qtf(w) \cdot \log\left(\frac{\mathrm{P}_{\mathrm{seen}}(w|\theta_{node})}{\mathrm{P}_{\mathrm{unseen}}(w|\theta_{node})}\right)$$

and the list is ordered by increasing node id.

5.2.2 Sum Node

The sum node maintains an array of result lists, with one result list for each of the children. It seeks to the next entry in each of the child result lists where the document id is at least as large as the requested document. If necessary, it calls the children nodes to get their next result lists. For the requested document, the sum node merges results from the result lists of the children, setting the score of the new result equal to the sum of the children's results with the same document and node id. This node assumes that results in a result list are ordered by increasing document id, then increasing node id. The results returned by this component have the score

$$\sum_{w \in Q, node} qtf(w) \log\left(\frac{\mathrm{P}_{\mathrm{seen}}(w|\theta_{node})}{\mathrm{P}_{\mathrm{unseen}}(w|\theta_{node})}\right)$$

and the minimum document id returned by the children is returned.

5.2.3 Score Adjustment Node

The score adjustment node adds

$$\sum_{w \in Q} qtf(w) \log \mathrm{P}_{\mathrm{unseen}}(w|\theta_{node})$$

to each of the results, where

$$\mathrm{P}_{\mathrm{unseen}}(w|\theta_{node}) = unseen(w, node)$$

as defined for the term node. If there is a prior probability for the node, the score adjustment node also adds on the log of the prior. The results in the list now have the score

$$\sum_{w \in Q, node} qtf(w) \log\left(\frac{\mathrm{P}_{\mathrm{seen}}(w|\theta_{node})}{\mathrm{P}_{\mathrm{unseen}}(w|\theta_{node})}\right) + \sum_{w \in Q} qtf(w) \log \mathrm{P}_{\mathrm{unseen}}(w|\theta_{node}) + \log(\mathrm{P}(node))$$

$$= \log(\mathrm{P}(\mathrm{Q}|\theta_{node})\mathrm{P}(node))$$

which is the log of the score by which we wish to rank document components.

5.2.4 Heap Node

The heap node repeatedly calls its child node for result lists until the document collection has been ranked. The next document id it calls for its child to process is the document id returned by the child node in the previous evaluation call. It maintains a heap of the top *n* results. After the document collection has been ranked, it sorts the results by decreasing score and stores them in a result list that is returned.

5.2.5 Other Nodes

There are many other useful nodes that could be useful for retrieval. One example is a node that filters the result lists so that the XML path of the node in the document tree satisfies some requirements. Another example is a node that throws out all but the top *n* components of a document.

6. EXPERIMENT SETUP

The index we created used the Krovetz stemmer and InQuery stopword list. Topics are similarly processed, and all of our queries are constructed from the title, description, and keywords fields. All words in the title, description, and keywords fields of the topic are given equal weight in the query. Table 3 shows the size of components created to support retrieval on the INEX document collection. The total index size including information needed to do context sensitive smoothing is about 70% the size of the original document collection. A better compression ratio could be achieved by compression of the context sensitive smoothing support files. Note that the document term file which is 100 MB is not necessary for the retrieval algorithms described above.

Component	Size (MB)
Inverted file	100
Document term file (allows iteration over terms in a document)	100
Document structure	30
Attributes inverted file	23
Smoothing – single collection model	4
Smoothing – context sensitive models (not compressed)	81
Other files (lookup tables, vocabulary, table of contents, etc.)	12
Total	**350**

Table 3: Lemur structured index component sizes

Topic Fields	Context	Prior	Path	inex_eval Strict	inex_eval Gen
TDK	YES	NO	NO	.0464	.0646
TDK	YES	YES	NO	.0488	.0653
TDK	NO	NO	NO	.0463	.0641
TDK	NO	YES	NO	.0485	.0654

Table 1: Performance of the retrieval system on INEX 2002 CO topics. Context refers to context sensitive smoothing, prior refers to the document component type priors, and path refers to the overlapping path filter.

Run Name (Official runs are bold)	Topic Fields	Context	Prior	Path	inex_eval Strict	inex_eval Gen	inex_eval_ng Strict	inex_eval_ng Gen	w/o overlap Strict	w/o overlap Gen
LM_context_TDK	TDK	YES	NO	NO	.0717	.0804	.2585	.3199	.2305	.2773
LM_context_typr_TDK	TDK	YES	YES	NO	.0769	.0855				
LM_context_typr_path_TDK	TDK	YES	YES	YES	.0203	.0240				
LM_base_TDK	TDK	NO	NO	NO	.0783	.0861				
LM_base_typr_TDK	TDK	NO	YES	NO	.0764	.0847				
LM_base_typr_path_TDK	TDK	NO	YES	YES	.0204	.0234				

Table 2: Summary of runs and results for INEX 2003 CO topics.

Table 4 shows approximate running times for index construction and retrieval. The retrieval time for context insensitive smoothing is reasonable at less than 20 seconds per query, but we would like to lower the average query time even more. We feel we can do this with some simple data structure optimizations that will increase memory reuse.

Action	Time (mins)
Indexing	25
Retrieval of 36 INEX 2003 CO topics – context insensitive smoothing	10
Retrieval of 36 INEX 2003 CO topics – context sensitive smoothing	45

Table 4: Indexing and retrieval times using Lemur

The higher retrieval time for the context sensitive retrieval algorithm is due to the recursive computation of the *unseen* component of the score as described Step 21 of Section 5.2.1. Clever redesign of the algorithm may reduce the time some. However, all of the descendent nodes in the document's tree must be visited regardless of whether the descendent nodes contain any of the query terms. This means that the computation of the *unseen* component of the scores is linear in the number of nodes in the document tree, rather than the typically sub-linear case for computation of the *seen* score components. If the λ and ω functions and their parameters are known, it is possible to precompute and store necessary information to reduce the running time to something only slightly larger than the context insensitive version. However, our implementation is meant for research, so we prefer that these parameters remain easily changeable.

7. EXPERIMENT RESULTS

We submitted three official runs as described in Table 2. All of our runs used the title, description, and keyword fields of the topics. Unfortunately, two of our runs performed rather poorly. This is either an error in our path filter or a problem with the component type priors. We would also like to evaluate the additional runs corresponding to the dashes in the table, but we have not been able to do these experiments yet.

The LM_context_TDK run has good performance across all measures. This is our basic language modeling system using context sensitive smoothing. The strong performance of the context sensitive language modeling approach speaks well for the flexibility of language modeling.

For the content only topics, context sensitive smoothing does not help. The node type priors also do not consistently help. There was a significant problem with the path filters we used.

With regards to context sensitive smoothing, it may not make much difference for content only tasks as they are typically searching for textual components such as paragraphs, sections, and articles. The characteristics of the text in these components tend to be very similar, so the context sensitive smoothing may not be helpful.

With regards to component type priors, we have observed similar puzzling behavior in [12]. We discovered that the distributions observed in the rankings after applying the prior probabilities are not the desired distributions. We are actively working on new techniques to incorporate information in a way that will provide the desired distributions of results in the rankings.

8. RELATED WORK

There exists a large and growing body of work in retrieving information from XML documents. Some work is described in our previous paper [11] and much of the more recent work is also described in the INEX 2002 proceedings [14]. With that in mind, we will focus our discussion of related work on language modeling approaches for structured document retrieval.

In [5] a generative language modeling approach for content only queries is described where a document component's language model is estimated by taking a linear interpolation of the maximum likelihood model from the text of the node and its ancestors and a collection model. This corresponds to a special case of our approach. Our model is more flexible in that it allows context sensitive smoothing and different weighting of text in children nodes.

The authors of [9] also present a generative language model for content only queries in structured document retrieval. They estimate the collection model in a different way, using document frequencies instead of collection term frequencies. As with [5], this model can be viewed as a special case of the language modeling approach presented here.

9. CLOSING REMARKS

We presented experiments using a hierarchical language model. The strong performance of language modeling algorithms demonstrates the flexibility and ease of adapting language models to the problem. In our preliminary experiments with standard text queries, context sensitive smoothing did not give much different performance than using a single collection model.

We described data structures and retrieval algorithms to support retrieval of arbitrary XML document components within the Lemur toolkit. We are reasonably pleased with the efficiency of the algorithms for a research system, but we will strive to improve the algorithms and data structures to reduce retrieval times even further.

In our future work, we would like to compare the component retrieval to standard document retrieval. We would also like to investigate query expansion using XML document components. Additionally, we would like to explore different ways of setting the λ weights on the nodes' language models, as we believe that words in some components may convey more useful information than words in other components.

10. ACKNOWLEDGMENTS

This work was sponsored by the Advanced Research and Development Activity in Information Technology (ARDA) under its Statistical Language Modeling for Information Retrieval Research Program. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors, and do not necessarily reflect those of the sponsor.

11. REFERENCES

[1] Fuhr, N. and K. Großjohann. XIRQL: A query language for information retrieval in XML documents. In *Proceedings of the 24th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval* (2001), ACM Press, 172-180.

[2] Grabs, T. and H.J. Schek. Generating vector spaces on-the-fly for flexible XML retrieval. In *Proceedings of the 25th Annual International ACM SIGIR Workshop on XML Information Retrieval* (2002), ACM.

[3] Hatanao, K., H. Kinutani, M. Yoshikawa, and S. Uemura. Information retrieval system for XML documents. In *Proceedings of Database and Expert Systems Applications* (DEXA 2002), Springer, 758-767.

[4] Hiemstra, D. *Using language models for information retrieval*, Ph.D. Thesis (2001), University of Twente.

[5] Hiemstra, D. A database approach to context-based XML retrieval. In [14], 111-118.

[6] Kazai, G., M. Lalmas, and T. Rölleke. A model for the representation and focused retrieval of structured documents based on fuzzy aggregation. In *The 8th Symposium on String Processing and Information Retrieval* (SPIRE 2001), IEEE, 123-135.

[7] Kazai, G., M. Lalmas, and T. Rölleke. Focussed Structured Document Retrieval. In *Proceedings of the 9th Symposium on String Processing and Information Retrieval* (SPIRE 2002), Springer, 241-247.

[8] Lafferty, J., and C. Zhai. Document language models, query models, and risk minimization for information retrieval. In *Proceedings of the 24th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval* (2001), ACM Press, 111-119.

[9] List, J., and A.P. de Vries. CWI at INEX 2002. In [14], 133-140.

[10] Myaeng, S.H., D.H. Jang, M.S. Kim, and Z.C. Zhoo. A flexible model for retrieval of SGML documents. In *Proceedings of the 21st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval* (1998), ACM Press, 138-145.

[11] Ogilvie, P. and J. Callan. Language models and structured document retrieval. In [14], 33-40.

[12] Ogilvie, P. and J. Callan. Combining Structural Information and the Use of Priors in Mixed Named-Page and Homepage Finding. To appear in *Proceedings of the Twelfth Text REtrieval Conference (TREC 2003)*.

[13] Ponte, J., and W.B. Croft. A language modeling approach to information retrieval. In *Proceedings of the 21st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval* (1998), ACM Press, 275-281.

[14] Proceedings of the First Workshop of the Initiative for the Evaluation of XML Retrieval (INEX). 2003, DELOS.

[15] Westerweld, T., W. Kraaj, and D. Heimstra. Retrieving web pages using content, links, URLs, and anchors. In *Proceedings of the Tenth Text Retrieval Conference, TREC 2001*, NIST Special publication 500-250 (2002), 663-672.

[16] Zhai, C. and J. Lafferty. A study of smoothing methods for language models applied to ad hoc information retrieval. In *Proceedings of the 24th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval* (2001), ACM Press, 334-342.