A Description of the LAMB Web-Derived Language Model Builder

Edward K. O'Neil James C. French*
Department of Computer Science
University of Virginia
Technical Report CS-2000-31

May 16, 2000

Abstract

This paper describes the language modeling script constructed for the Spring 2000 Information Retrieval seminar. The script builds on Callan's[1] work of automatically generating language models for text databases by creating language models for internet search engines. An overview of the work, principles and observed properties of language models, the language modeling script (LAMB) and its shortcomings are described.

1 Introduction

The Spring 2000 Information Retrieval (IR) seminar has focused on IR concepts in general as well as distributed information retrieval. One of the major topics of the seminar focused on efficient and effective metasearching. If collection selection is employed in distributed searching, a proxy form of the sub-collections is needed to represent the real collection at query time. We refer to this proxy form as a language model of the collection. The interest of this paper is on how to efficiently and automatically create a language model that accurately reflects the collection's contents.

In cooperative distributed search environments, language models are obtained from each of the collections in the federation of sources. Then a server is able to examine all models of the collections at query time, and the query is issued to the appropriate collections. Problems may arise from this method because of different indexing, stopping, or stemming strategies at each site, but the sites cooperate with the server to provide information about their contents. Obviously, this method is not effective for sites that do not voluntarily provide a language model representing their contents. Callan et al.[1] present the query-based sampling method as a way to subvert uncooperative collections by automatically learning the language model of a given site[1]. This process proceeds by approximate random sampling at the source via a query interface and construction of the language model from the documents returned from an issued query[1]. Their query-based sampling approach showed that an accurate language model representing the contents of a collection can be learned automatically by analyzing documents returned from a small number of queries posed to the site.

Our interest focuses on internet search engines. Since the inception of search engines in 1994, many search sites have transformed themselves to become internet "portals" where information / products may be brokered in many forms. In recent years, category structures have appeared at many of the major search engines. The Open Directory Project (ODP)[1] is a worldwide effort

*french@cs.virginia.edu

[1]http://www.dmoz.org

Altavista	Infoseek	Lycos	NorthernLight	Snap	Yahoo
(16)	(19)	(15)	(16)	(22)	(14)
arts	arts & humanities	arts & entertainment	arts	arts & humanities	arts
autos	automotive	autos		autos	
business & finance	business	business & careers	business & investing	business	business & economy
	careers			careers	
computers	computing	computers & internet	computing & internet	computing	computers & internet
			contemporary life		
	education		education	education	education
	entertainment		entertainment	entertainment	entertainment
	family			family & kids	
				finance	
games		games		games	
	gov't & politics		gov't, law & politics		government
health & fitness	health	health	health & medicine	health	health
home & family		home			
			humanities		
internet					
	living			lifestyle	
				local	
	marketplace				
	money				
				multimedia	
news & media	news	news		news	news & media
	people				
				politics	
			products & services		
	real estate				
recreation & travel		recreation			recreation & sports
reference		reference	reference	reference	reference
regional		regional			regional
				retailers	
science	science	science & tech.	science & mathematics	science & tech.	science
			social sciences		social science
society & culture		society & culture			society & culture
sports	sports	sports	sports & recreation	sports	
			technology		
				teens	
	travel	travel	travel	travel	
world					

Table 1: Category structure of six search engines in Spring 2000.

to create a categorically structured set of documents that is available for free on the internet. This category structure includes topics ranging from automobiles and computers to teens and the Greek art. Many search engines have based their category structures on this classification including AltaVista, Lycos, and Google. Others such as Yahoo have taken up the effort independently to create a classification structure of web pages, and yet another site, Northern Light, seems to automate this classification of pages using proprietary algorithms. We have used the structure that the categorized search engines are providing as a natural decomposition for creating collections[4], breaking them apart into one collection per top level category. For example, in the spring of 2000, AltaVista provides 15 categories in its Directory (shopping is excluded because of unique properties arising when searching shopping sites). We would consider each of these 15 categories to be separate collections. Our goal is to obtain a representative and accurate language model for each of these collections and have initially done this for six internet search engines and their top level categories including AltaVista, Infoseek, Lycos, Northern Light, Snap, and Yahoo. This decomposition currently consists of 102 different categories or collections as shown in Table1.

We have obtained these language models using Callan's methodology of query-based sampling. The work conducted in this project applies their findings in the form of a Perl script that automatically generates a language model for a whole search engine or a search engine category structure using a small amount of search engine specific information (see Section 4). Properties of language models are discussed in the following.

2 Language models

A language model is a representation of the content of an information source. The traditional IR model of an inverted list would be a language model since it describes the content of the source by representing the document frequencies of each term appearing in documents found in the collection. Ideally, models will consist of the exact terms in their exact counts of occurrence in the engine's category itself. Realistically, this is unlikely because the search engines are not keen on the idea of turning out a list of the pages they have indexed. In order to construct the complete language model via sampling, every page would have to be found in a sample, and using our sampling strategy, each page would have to appear in the top k results (that we view) of some query. This requirement may not even be possible if certain pages never appear above the rank $k + 1$ for any query. Sampling will approximate the result of the correct (complete) language model and will provide a snapshot of the pages the engine indexes. Initially, we have started off with an almost random sampling strategy, the implications of which will be discussed later.

Our choice of language model has a very simple structure, but its construction and components are highly dynamic and complex. Our models consist of term tuples, the stopped/stemmed term, document frequency, and collection term frequency. The first two are of the most interest and utility. The construction of the model proceeds as follows; the general process is that described by Callan et al.[1]. Initially, a seed query retrieves a set of documents, and these documents contribute a set of vocabulary to the empty language model. From the unstopped, unstemmed terms, the next seed query is selected. The choice of the query term has implications on the outcome of the language model (see section Query Term Selection). Regardless of how the next term is chosen, it will reflect biases in the pages seen so far. For example, if a set of German pages are brought back as a result of a query "Porsche" in an automotive category, the language model now has a slight hint of German language pages, increasing the chances of seeing more similar pages in the future, and this is true of any topic. Sampling strategies will need to move between sub-topics within a category to capture the whole content of a category. Feeding only at the most frequently occurring documents does not include the more infrequent / obscure content from the collection.

We have chosen generic seed queries in an attempt to initially retrieve a set of random documents hopefully not biased toward any one concept; the terms rock, being, and apple have been used. Obviously, these terms have different meanings in different categories, but the hope is that they are generally generic. Callan notes, however, that the selection of this term has little effect on the end language model[1]. After the first pages are retrieved, language modeling will hopefully jump between topic clusters within a collection. Think of this this as moving between sub-categories within a root category at a search engine. Hopefully, enough general terms will be added that this motion does not require a term occurring infrequently in the true language model to access some cluster of similar pages, and the ratio of visited to unvisited sub-categories is high.

2.1 Difficulties with web data

This strategy of polling on the wild and wooley web has its difficulties, though. One problem we encountered was that of content length; the range of document sizes is extreme, from megabytes to under fifty bytes. The problem that arises is that of biasing the language model to the larger content. Before using document type as a decision criterion for fetching a page, we were fetching text files from Project Gutenberg.[2] A query in AltaVista arts would bring back results with several classic literature works (as large text files) in the top ten hits. The text of Moby Dick adds a significant

[2]A source of many electronic texts. See http://sailor.gutenberg.org/.

block of many English language terms to the language model. This pollutes the model because it might not reflect the content that appears in the category in an accurate manner. Paradoxically, another problem was short documents. For example, a document not found error (404 Error) or root frame page would bring back a hit for a page but would not contribute any useful content. Both of these experimental roadbumps caused us to implement size ranges on the documents we retrieved. These limits are variable but are currently set to a minimum size of 100 bytes and a maximum of 100K bytes. We also fall prey to the problem search engines initially encountered that page creators can trick us into believing their page is related to a given topic by repeating a term hundreds of times. We have encountered this problem infrequently, and do not account for it currently. Similarly, the problem of dead links influences the execution time of the modeler and the number of hits returned per query. Without formalized stopping criteria (see section 4.2), we are using a target number of pages to retrieve from a category, ranging anywhere from 300 to 1500. Informal experiments show that in the average category over all the page requests made (obviously, the number of requests is larger than the target number of pages), about 5% of the total links returned are dead and about 8% of the links returned are too short. We do not count these pages toward the target number of pages, and often, the number of page requests exceeds the target number of pages by 25% to 50%. The text document problem also prompted us to put limits on the types of documents that we gather. Modeling runs have been limited to web pages that are most likely html files while excluding text, proprietary binary format (Word documents, PDF/Postscript files), images, animations, programs, and other document types. This keeps the content focused on accessible HTML pages.

Callan[1] notes several benefits of this type of language modeling that we also realize here. The representation of terms across all models is consistent because the same stoplist and stemmer are used for all terms. Thus, no vocabulary resolution is necessary. Also, we have been able to model the languages of rather uncooperative search engines by using the engine's own interface to sample with queries.

3 Language Modeler

3.1 Approach

The language modeler is a Perl script that is seeded with a total number of pages to see, an initial one-term query, and a configuration file. The script relies on several Perl modules for its web abilities, the LWP::UserAgent and URI::URL models. Otherwise, the script lives independently of third party tools. The script implements its own stopping algorithm using a 421 word stoplist and uses a Porter stemmer[3] implemented from the algorithm presented in Frakes[2]. Configuration files and scripts have been implemented for each of the six search engines in Table 1 (AltaVista, Infoseek, Lycos, Northern Light, Snap, and Yahoo) The following discusses specific implementation details.

Given a query and engine to poll, the first step is to submit the query to the search engine and retrieve the results. In our sampling runs to date, a total of up to ten pages are viewed for each query. Each page is retrieved using the URL the search engine returned, the HTML tags/code are removed to the best of our ability, stopwords on the page are removed, the remaining content is stemmed, and the resulting terms are then added to an accumulating language model of the contents of the search engine category being sampled. This process proceeds until a set number of pages have been seen. Dead links and obviously non-HTML documents are dropped from the list

[3]Available at ftp://ftp.vt.edu/pub/reuse/IR.code/ir-code/stemmer.

of pages seen for each query; usually less than the top N results from a query are actually useful because a link may be dead, too short (long HTML documents are truncated to a byte length), or not HTML content.

Once all of the pages that can be viewed for a query out of the up to top 10 results have been fetched, parsed, and included in the model, another query term is selected. All query terms are single term queries, and currently, this term is taken from a growing list of the total vocabulary (stopped but unstemmed) seen so far. The term is chosen by picking a random number that is seeded from the CPU clock and indexing into the key array of this hashtable. Duplicates are not allowed, so terms are stored exactly once. The resulting word is used as the next query, and the process described above is applied until seeing the target number of pages. Also, a flag denoting whether this term has been used as a query is set so it is not chosen again, that is, we sample the terms without replacement.

The hypothesis is that by randomly sampling a category's pages, the language modeler will, over time, gain a representative view of the vocabulary content contained in the category.[4]. The resulting language models for each category will represent the collection of documents during query processing when they will be evaluated against each other relative to a given user query. If we can effectively create language models of categories by sampling in this way, we can use a small representation of the category's pages to build an accurate language model of the collection. In addition, we can construct a metasearcher that analyzes these underlying category representations versus a query and uses a selection algorithm to choose the most effective set of subcollections to which to send a query.

3.2 Parameters

Three parameters are required to execute the script:

- configuration file;
- number of pages to see; and
- seed query.

Optional parameters include:

- target directories for the output;
- paranoid mode flag; and
- command line specification of a single category to sample.

3.3 Search engine specifics

The script has several subroutines that must be tailored to meet the format of each search engine. First, the page parser, getSearchPageResultURLs(), returns the list of links that are the results of the search and that appear embedded in the HTML. A Perl module exists to perform this operation, but it is not used for two reasons. First, the Perl module that yanks URLs from an HTML document is slow, and second, the results of the search are not the only links that appear on a search results page. Many dozens of other links to advertisers and merchants appear intermixed

[4]In other work we are examining some different statistics that can be used for database selection in the WWW environment[5] and that can be used as the basis for a stopping criterion for the modeler[3]

with the search results. Still, we have not found a search engine that can not be exploited simply by examining the structure of the results page, and there has always been a unique marker of some sort that delimits the start and end of the search pages.

Another engine specific characteristic is that each search engine has a different URL syntax for submitting the search request. Usually, this is done through CGI and uses both visible and hidden parameters that are passed to the engines. Some search engines such as AltaVista require only the query and category identifier (a number or a string) to be embedded in the URL in order to successfully complete a query in a specific category. Others engines such as Snap or Yahoo require multiple category identifiers to successfully complete a query. These differences are reflected in the configuration file that is passed to the script at runtime. These parameters must be parsed differently for each engine from the configuration file. A multi-dimensional hashtable is provided for storage of variables used to specify categories. Each category's configuration data should be inserted into the hashtable and keyed with some kind of unique category identifier that is taken from the configuration file. In automatic mode, the script will simply run through each element in the hashtable and model the language of the category represented by the data stored at that element. The subroutine to be modified is called readConfigFile() and the hashtable for storing the is called 'configInfo.'

The final engine specific characteristic is how the query URLs are built. Each configuration file provides a search URL template that is specific to each engine and has its field identifiers embedded within it. For example, AltaVista appears as:

```
http://www.altavista.com/cgi-bin/query?pg=q&q=<query>&crid=<category>&kl=en&stype=dptext
```

and at run-time, the <query> and <category> tags are replaced with values for the current query and the AltaVista specific category identifier. Engines such as Snap and Yahoo that vary multiple fields per category will have additional named fields to be replaced here. This structure will also differ when considering a search engine in total or a ctegory in its directory structure.

3.4 Pages to see

The number of pages to process parameter is controversial and undesired. Currently, the language modeler samples a resource until it has seen a set number of pages; however, the number of pages necessary to capture an accurate model of the language of a resource may be much higher or much lower. The problem is discovering what quantitative characteristics of a language model imply that the model has converged on a stable set of vocabulary that represents the content of the collection. The speed at which this happens may be influenced by factors such as the collection size, average document size, diversity of the collection, and choice of the next query. Our hypothesis is that a stable set of core vocabulary will accumulate in the language model consisting of terms with document frequence greater than one, and once this set has been reached, additional iterations on the model will contribute slowly and in two classes:

- terms that increment the DFs of the core terms
- additional "one-off" vocabulary (DF = 1 terms that remain DF = 1) terms that accumulate

Our goal of determining a stopping criterion will attempt to find the point at which the "core" vocabulary stabilizes.

3.5 Automatic/manual execution

In an effort to implement automatic execution but support testing and user needs, the script can run in two modes. One is a user specified mode where the user can provide a unique category key, target directory for output files, and number of pages to see. More frequently, the script will be run without a specific category and will be set loose to feed at a search engine for:

<number of pages> * <number of categories>

of pages to be seen; remember that the number of actual page requests may be much higher than this because of dead links, short pages, etc. Output is directed to a hierarchy whose directory names are constructed using the date of execution, number of pages seen, search engine name abbreviation, and unique category identifier.

3.6 Output files

Inside each directory (representing a category) created by the script, a set of files is produced. Each filename begins with a unique two letter abbreviation that identifies the search engine that produced the result. This will make pooling results from different engine's categories easier if necessary.

- <two letter abbreviation>.history – A history file is kept that keeps track of the entire query/response interaction between search engine and fetch/response interaction between the script and the target pages. The current query and number of pages seen so far are noted for each query. The list of URLs is then produced for each result the engine returned. The result of the page fetch is also noted as too small, too large, wrong type, or dead link. A successful page fetch has none of these identifiers.

- <two letter abbreviation>.queries – The queries file simply keeps a list of the queries that were issued and their order.

- <two letter abbreviation>.paranoid – Paranoid mode is run when it is of interest to keep track of the total vocabulary that was contributed by each page delimited by which pages contributed the terms. Using this flag (-p) reduces the efficiency of execution slightly because the vocabulary of a given page (up to the maximum page size) is written to the file. This data has been useful for debugging, determining the language of a page, and finding where some of the bizarre terms in the model originate.

- <two letter abbreviation>.lm.<category ID>.<total pages seen>.<number of queries issued> – The language model files are built as the script executes and are saved incrementally while advancing toward the total number of pages to see. Language models can be created once per category, or they can be created incrementally as the model is accumulating at any granularity. This feature, incremental language models, provides the ability to watch the language models evolve and has great significance for analyzing the stopping criterion of a model. By keeping track of the language models at a granularity down to every page seen, we can examine the number and characteristics of terms added at every step. Hopefully, this information will help determine what characteristics help produce a high-fidelity language model.

3.7 Query execution

As has been noted, queries are chosen randomly in the current script from the accumulated (unstemmed) vocabulary. In using web data/queries as opposed to those from TREC, several wrinkles arise. For every query chosen and a target of N documents to see, the total number of documents that may be seen can range from 0 to N. This is not a problem but a bump that must be remembered when performing analysis.

The modeler must also keep track of pages that have been seen in a running history list. The history list is checked before a page is fetched to make sure that it has not been seen previously. Allowing duplicate pages in the language model does not accurately reflect the content of a collection because it over-represents the constituent parts of the pages in the model. This process is done on a best effort basis; the DNS is not queried to check for hostname aliases; the URL of the current page is simply compared against the URL of every previous page. If a match is found, the page is not fetched. Catching all aliased pages would be extremely difficult because of nested directory hierarchies and other strange (ie real world) web site structures. The only way to prevent duplicating pages would be to store and compare the content of every page against every other page's content each time a page is fetched.

3.8 Summary

The modeler collects a fixed number of pages from a collection. The resulting language model is a best effort attempt to create an accurate representation of the content of the underlying collection. Stopped/stemmed terms are included with their document and term frequencies, and several output files tell the epic saga of the harvesting of pages that are stored in a cohesive and well-defined hierarchy of directories.

4 Shortcomings of the Modeler

The modeler performs a very useful task in generating the language models of search engine document collections; however, it has several operational shortcomings that affect the end language models.

The modeler makes the significant assumption that HTML existing on the web is standards compliant. This is specifically important when removing the HTML features from a web page in an effort to end up with the terms of the page content itself. It must account for all forms of mark-up and scripting languages and attempts to do so by removing a set of known tags that denote this page content. If HTML is malformed and has dangling tags or tables, unwanted strings from these pages may be included in or terms may be excluded from the model. For example, this may result in Javascript method invocations or "br" terms showing up in the model. With the exception of excluding content (which has not been observed), the impact of this is annoying but negligible. In the end, the terms that are included in the model because of garbled HTML are infrequent both in DF count and in occurrence.

Also, the modeler does not exclude every non-HTML content type. Because we can not predict which pages are HTML content, we also can not predict which pages are not. The script makes a best effort to capture this content without performing Herculean analysis to detect duplicates. Dynamic content, such as active server pages and PHP pages, is not excluded.

Dead links are a nuisance but more of a shortcoming of the underlying search engine than the sampling strategy. The script will not include any page in the model that takes longer than sixty seconds to reply. Clearly, if the network fails while a script is running, the model will not be regarded as complete.

4.1 Query term selection

The two most significant shortcoming of the language modeler are discussed starting with the selection of the next query term. Using a random number seeded with the clock seems, in informal experiments, to produce diverging language models (even when seeded with the same query). Initial studies into the effects of query term selection indicate that how a term is selected, but not what a term is, has a significant impact on the characteristics (size, df count) of the resulting model. Modifying the script to include more significant and principled query selection techniques would not be a difficult task and benefits to language model fidelity may be significant; however, one must be uncovered first.

We make an effort to randomly select a term and do not duplicate the use of that term in a query. Callan investigates several criteria for term selection including attempting to prevent selection of a term that would return few or no results[1]. We make no such guarantee because we can not know the underlying content of the search engines, how many documents will return for a query, and how many of the returned links are valid. Additionally, Callan's work requires non-numeric terms of length greater than three. He also investigates whether to select terms from another language model (olm) or the current learned language model (llm), but we have not performed such experiments. They demonstrate that random term selection from the llm is more effective than selecting from the frequent terms[1] which has been our approach also. Our term selection methods have been:

- random selection (seeded with the clock); and
- random selection (seeded with the clock) excluding the df=1 terms.

The current hypothesis is that effective query term selection throughout the harvesting process will speed convergence and increase the quality of the language model.

4.2 Stopping Criterion

Another issue with the approach developed in the script is that it does not have an automatic stopping criterion. As mentioned before, the modeler runs until a total number of pages have been sampled. Currently, we are guessing and trying to over guess the number of pages that must be seen to reach an accurate language model. A more effective solution would be to develop an automatic stopping criterion that halts the harvesting of pages when the fidelity of the language model reaches certain statistical properties. Currently, we are speculating that this criterion may be the point at which the accumulation of terms with DF equal to one settles down and starts to steadily increase. Alternately, it may occur as terms "freeze" into ratios constrained to a range centered around their ratios in the hypothetical final language model. Terms that do freeze would have to do so and they stay at the same ratio for a set amount of number of pages seen. A preliminary study can be found in [3]; these investigations are ongoing.

5 Shortcomings of search engines

During the course of careful study of several search engines, several shortcomings of their operation have also bee noted. Since these experiments are performed on raw web data, the results are expected to reflect the nature of this real world data source; however, the search engines themselves provide an entertaining variable in the mix.

The changes in the underlying search engine page formats is a certainty. This means that the way in which data is laid out on a page is very dynamic in nature and that engine specific parsing code is likely to become obsolete over time. These Perl subroutines are likely to have to be rewritten (in addition to the configuration files). This has already happened with AltaVista.

Construction of a language model using query-based sampling is also constrained by the performance / algorithms of the underlying engine. For example, AltaVista has a tendency to return garbage results to simple queries and, in experiments with two term queries, does not handle quoted phrases appropriately at all times. In addition, we have sometimes noticed that terms contributed by a page from an AltaVista category to an evolving language model are not queryable, ie they return zero results although they are not common stopwords and do appear in indexed documents. We speculate this to be an artifact of AltaVista indexing only the top N bytes of a page where N is less than the maximum number of bytes that we index (which begs the question of whether our models actually more accurately represent AltaVista's content than their own). Also, several search engines use the ODP list as a basis for their category structure. One that does not is Northern Light which uses its own categorization hierarchy. Experiments with the Northern Light engine have returned categories that are much broader in defining which pages are relevant to a given category. Examples of this behavior include querying on "Java" when restricted to the arts category. The results include pages pertinent to Java animation applets, which could loosely be defined to be relevant to "arts" because they mention animation; however, their presence in the category tends to indicate differences in the meaning of a Northern Light versus an AltaVista category. Given this behavior and the fact that Northern Light has job postings for automatic classifier software engineers, Northern Light may be doing automatic page classification which would make its language models different from models of collections built by people. Search engines also have an unpredictable rate of refresh in their category hierarchies. While initially investigating this process, Google grew categories with a structure that was obviously fresher than that provided at AltaVista. While AltaVista simply may choose not to provide pages on new music groups, movies, or computer technologies, in all likelihood, AltaVista had not updated its directory structure recently at the time we viewed the contents. These are some of the difficulties and surprises when working with real web data.

6 Future work

The two most important issues to be addressed in future work are the significant shortcomings of the script, the query selection algorithm and a stopping criterion. Both of these features when discovered and working properly should help boost the speed and fidelity with which language models can be built by query-based sampling. Work conducted in these two areas will build on the experiments in stopping criteria and term selection that Callan initiated in 1999.

Structurally, the script would benefit from several changes including a modularization of the operations that interact with the search engines and allowing for restart. The subroutines that interact with the engines should be inserted into a Perl module that encapsulates the engine specific routines and would use function pointers to access them when modeling a specific engine. Additionally, we have discussed the ability to restart a previous run from a history of pages, current language model, and query list; however, the necessity of this feature is unclear. Obviously, implementation of stopping criterion and query term selection would be necessary and could be easily bolted on to the existing machinery.

The construction of 102 language models will provide the content representation on which to base a metasearcher that employs collection selection. This search tool will perform collection selection to determine the most appropriate collections at which to pose a given query. Accurate language models will yield a more effective search result by representing the content of the underlying engines in small, automatically generated language models. These models can be updated as necessary, using this script, to reflect changes in the collections and improve metasearch results.

7 Conclusions

This project has been a good study of the process that must go into creating a query-based sampling language modeler that harvests web data from search engines. Many intangible lessons have come out of interacting with search engines and real world data, and several concrete observations of the traits of search engines, processing web pages, and language models have been noted.

A script has been constructed that samples a collection automatically and incrementally accumulates terms in language models. Future work will refine this process to be a more principled gathering of content, resulting in higher fidelity models. These models will be useful for implementing a metasearcher that selects the most appropriate search engine collections to which to pose a query. This activity, termed fine-grained metasearching, is being explored more fully by Powell[4].

References

[1] Jamie Callan, Margaret Connell, and Aiqun Du. Automatic discovery of language models for text databases. In *Proceedings of the ACM–SIGMOD International Conference on Management of Data*, pages 479–490, 1999.

[2] W. B. Frakes and R. Baeza-Yates. *Information Retrieval: Data Structures and Algorithms.* Prentice Hall, Englewood Cliffs, NJ, 1992.

[3] G. A. Monroe, D. R. Mikesell, and J. C. French. Determining stopping criteria in the generation of web-derived language models. Technical Report CS-2000-30, Department of Computer Science, University of Virginia, May 2000.

[4] Allison L. Powell. *Database Selection in Distributed Information Retrieval: A Study of Multi-Collection Information Retrieval.* PhD thesis, Department of Computer Science, University of Virginia, January 2001. Forthcoming.

[5] R. Srinivasa, T. Phan, N. Mohanraj, A. L. Powell, and J. C. French. Database selection using document and collection term frequencies. Technical Report CS-2000-32, Department of Computer Science, University of Virginia, May 2000.