# Fast Query Expansion Using Approximations of Relevance Models

Marc-Allen Cartright
CIIR
Dept. of Computer Science
UMass Amherst
Amherst, Massachusetts
irmarc@cs.umass.edu

James Allan
CIIR
Dept.of Computer Science
UMass Amherst
Amherst, Massachusetts
allan@cs.umass.edu

Victor Lavrenko
School of Informatics
University of Edinburgh
Edinburgh, UK
vlavrenk@inf.ed.ac.uk

Andrew McGregor
Dept. of Computer Science
UMass Amherst
Amherst, Massachusetts
mcgregor@cs.umass.edu

## ABSTRACT

Pseudo-relevance feedback (PRF) is a retrieval technique that improves search quality by expanding the query using terms from high-ranking documents. These approaches work by running the original query, analyzing some number of top documents, and then running a second query derived from that analysis. Although PRF can often result in large gains in effectiveness, it is rarely used in practical settings because running two queries is time consuming, particularly when the new query might be many times larger than the original.

We describe a PRF method that uses corpus pre-processing to achieve query-time speeds that are nearly the same as those of the original queries. Specifically, we show that a language modeling based PRF method, Relevance Modeling, can be recast to benefit substantially from a pre-processing step of finding pairwise document relationships. Using the resulting method, the Fast Relevance Model (fastRM), we are able to reduce the online retrieval time by several orders of magnitude while still obtaining considerably improved performance over the unexpanded query.

We further explore methods for reducing the time required by the off-line document comparison step, investigating a range of approaches for approximating the comparisons. Our results show that certain approximations can substantially reduce the pre-processing effort and still achieve significant improvements in retrieval performance with only a small loss in effectiveness.

**Categories and Subject Descriptors:** Foundation of Information Retrieval, IR Architectures, Scalability and Efficiency



| Collection | # docs | Ratio (RM/LM) | |
|---|---|---|---|
| | | MAP | Ret. Time |
| AP89 | 84,678 | 1.21 | 191.37 |
| WSJ | 173,252 | 1.28 | 160.63 |
| Robust05 | 1,033,461 | 1.38 | 430.88 |

**Table 1: RM is more effective but *much* slower than LM**

**General Terms:** relevance model, pseudo-relevance feedback, distributed computing, algorithms

## 1. INTRODUCTION

Lavrenko and Croft's Relevance Model [17] is a variation of pseudo-relevance feedback methods developed for the language modeling framework. The approach continues to match or beat other information retrieval techniques. However, like all pseudo-relevance feedback (PRF) methods, Relevance Model implementations typically slow down query processing time by several orders of magnitude, making them unsuitable for real-time retrieval settings. The standard formulation of these methods involves submitting an original query, using the resulting ranked list to perform weighted query expansion, and performing a second round of retrieval. The second query can consist of hundreds of terms, resulting in a slow—sometimes impressively slow—evaluation over the collection. To illustrate the problem, consider Table 1 which compares gains in effectiveness to increases in processing time across several collections, in both cases as a ratio of RM compared to standard query likelihood language modeling (LM).

Even for a small collection such as AP89, the original Relevance Model is nearly 200 times slower than the Language Model, while providing a 20% relative improvement. The tradeoff illustrated here is unacceptable for any realistic setting. Even in a research environment where low query-processing throughput can be tolerated, a significant amount of potentially useful research time can be lost merely performing runs to tune parameters of the model.

We look to alleviate this issue in this study. We show how the Relevance Model can be reformulated to perform

much of the computation offline, drastically reducing the impact on retrieval time. We further investigate techniques for reducing the time and space requirements of the offline computation while avoiding a significant negative impact on retrieval performance. Although our experiments and discussion focus on the Relevance Model, the ideas generalize to most forms of PRF and are an important step towards overcoming the inefficiency that often plagues the approach.

The rest of this paper proceeds as follows. We start by describing the Fast Relevence Model approach in Section 2 and then demonstrate empirically its performance advantages in Section 3. In Section 4 we discuss ways of further reducing the time needed for fastRM, either by reducing the amount of information stored (Section 4.2) or calculated (Section 4.3). We wrap up by presenting some related work in Section 5 and conclude in Section 6.

# 2. FAST RELEVANCE MODELS

Lavrenko and Allan first proposed Fast Relevance Models in a 2002 technical report [16]. We summarize the key points here for clarity.

Given a collection of documents $C$ and the vocabulary of terms $V$, we score a document $D \in C$ based on its cross-entropy from the unobserved relevance model $R$:

$$H(R\|D) = \sum_{t \in V} P(t|R) \log P'(t|D) \quad (1)$$

where $P'(t|D)$ indicates the smoothed probability of $t$ occurring in document $D$. In practice, we approximate $P(t|R)$ in Equation 1 by assuming that the query $Q = q_1 q_2 \dots q_i$ is a small sample generated from distribution $R$, the latent relevance space we want to model. We perform an initial retrieval to generate a ranking of documents based on this sample. We then use the top ranked documents to form ordered set $\mathcal{M}$, which, by extension, acts as a larger sample of $R$, which we use to better approximate $P(t|R)$:

$$P(t|R) \approx P(t|q_1 \dots q_i) = \sum_{M \in \mathcal{M}} P(t|M) P(M|q_1 \dots q_i) \quad (2)$$

Substituting Equation 2 into Equation 1, we can then rearrange the two finite sums to produce the cross-entropy between the model samples and the documents in the collection:

$$\begin{aligned} H(R\|D) &= \sum_{t \in V} P(t|R) \log P'(t|D) \\ &= \sum_{t \in V} \sum_{M \in \mathcal{M}} P(t|M) P(M|q_1 \dots q_k) \log P'(t|D) \\ &= \sum_{M \in \mathcal{M}} \sum_{t \in V} P(t|M) P(M|q_1 \dots q_k) \log P'(t|D) \\ &= \sum_{M \in \mathcal{M}} \left[ \sum_{t \in V} P(t|M) \log P'(t|D) \right] P(M|q_1 \dots q_k) \\ &= \sum_{M \in \mathcal{M}} H(M\|D) \times P(M|q_1 \dots q_k) \quad (3) \end{aligned}$$

The key observation here is that $H(M\|D)$ is independent of the query and thus can be computed off-line. At query time, the set $\mathcal{M}$ of models is determined using a typical query likelihood approach (recall that models here are documents) and final scores are calculated by merging the $H(M\|D)$ scores, weighted by the models' retrieval scores. That is, the expensive step of issuing an expanded query has been replaced by much faster table lookup and score merging.

## 2.1 Storing $H(M\|D)$

We have shown a derivation of RM that makes it more efficient if $H(M\|D)$ is calculated off-line. Logically, we will do that by creating a $|C| \times |C|$ matrix, $A$, that includes cross-entropy values for all pairs of documents in $C$. In fact, it will be useful to sort the rows of the matrix in decreasing order of cross-entropy, meaning we will actually store the values as illustrated in Figure 1. Here, $M$ denotes a document selected as a sample of the unobserved relevance model, and $D$ denotes a general document in the collection. Every entry in the matrix contains a tuple $\langle id, score \rangle$, where $id$ is the document id of $D$, and $score$ is the cross-entropy of $M$ and $D$, $H(M\|D)$. Every row corresponds to a document in the collection, therefore selecting row $i$ corresponds to setting document $i$ as $M$.

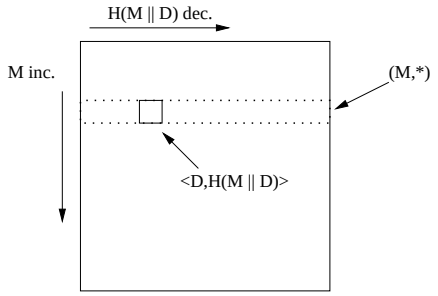


**Figure 1: A calculated matrix**

As mentioned earlier, the entries in each row are sorted in score-decreasing order. We use several compression techniques to reduce the storage requirements of the matrix. The document ids are stored using a zero-compressed encoding provided by the Hadoop library, and the scores are stored using a simplified version of a new DFCM compression scheme [22]. An index of the row positions is stored at the end of the matrix file. Upon opening, this index is read in once, so during retrieval moving to a row involves a single seek call to the matrix file. A *scan* of a row refers to repositioning to the row and iterating over the entries in a streaming fashion. Random accesses within a row are not supported.

## 2.2 Calculating $H(M\|D)$

Calculating cross-entropy between two documents lends itself nicely to the MapReduce framework, though some care must be taken to account for the smoothing factors involved in the second probability term. Consider $H(M\|D)$ again:

$$H(M||D) = \sum_{t \in M} P(t|M) \log P'(t|D) \quad (4)$$

At first glance, this formulation appears to fit neatly into a product-then-sum operation, which is ideal for a MapReduce environment. Determining an of inner product over a set of documents can be performed simply by incrementing a score accumulator using the posting lists of an indexed collection [10, 18]. This approach avoids having to send the content of every document to every node in the cluster in order to calculate its similarity with every other document. Intuitively,

it would seem that one could simply create the score increments by taking the cross-product of each posting list with itself, and emit each of the resulting increments from that matrix. After a simple shuffle and reduce, the scores would be produced. Our own postings list-based implementation, based on this process, is shown in Figure 2.

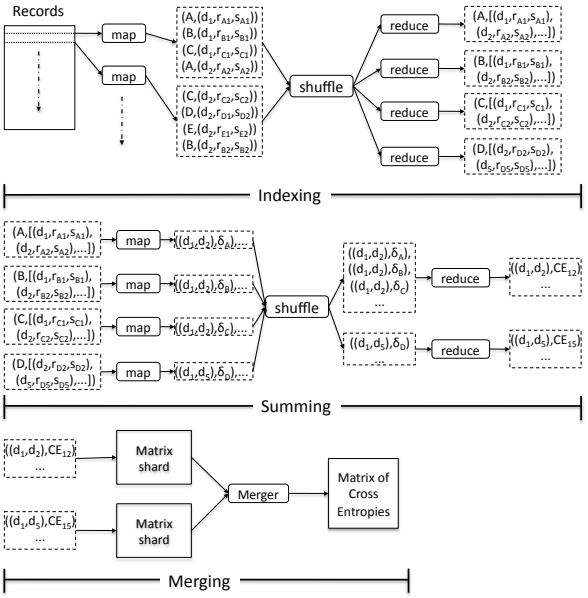


**Figure 2: The MapReduce implementation of matrix calculation.**

However, because of smoothing the situation is not so simple. Consider these two example documents:

$$d_1 = \{ \text{ The cat crossed the street } \}$$
$$d_2 = \{ \text{ The dog crossed the river } \}$$

From Equation 4:

$$\begin{aligned} H(d_1 \| d_2) = & P(\text{the}|d_1) \log P'(\text{the}|d_2) \\ & + P(\text{cat}|d_1) \log P'(\text{cat}|d_2) \\ & + P(\text{crossed}|d_1) \log P'(\text{crossed}|d_2) \\ & + P(\text{street}|d_1) \log P'(\text{street}|d_2) \end{aligned} \quad (5)$$

Notice that neither posting list for the terms "cat" nor "street" will have $d_2$ in them. While this is expected, it makes calculation of Equation 5 more complicated in MapReduce. We will never be able to generate the second nor the fourth additive terms of the sum in Equation 5, yet they are nonzero values, as the second part of each product is a *smoothed* probability, and therefore never zero.

We circumvent this issue by producing the background score for every document pair, and then incrementing from that point to produce the final scores. Let us begin with Equation 4 again:

$$H(M||D) = \sum_{t \in M} P(t|M) \log P'(t|D) \quad (6)$$

meaning if the term does not occur in $D$ but does occur in $M$, we still need to calculate the term's contribution to the document pair. Brief experiments conducted indicate that Jelinek-Mercer smoothing is superior to Dirichlet smoothing for this method, therefore we expand our smoothed probability as follows:

$$P'(t|D) = \lambda P(t|D) + (1 - \lambda) P(t|C) \quad (7)$$

Let $\beta(t)$ represent the background probability of $t$. This is the quantity $(1 - \lambda)P(t|C)$ in Eq. 7:

$$\sum_{t \in M} P(t|M) \log P'(t|D) = \sum_{t \in M} P(t|M) \log(\beta(t)) + \sum_{t \in M \cap D} P(t|M) \left[ \log(P'(t|D)) - \log(\beta(t)) \right]$$

Note that the sums are over different sets: $M$ or $M \cap D$.

We now focus our attention on the second term to expose the smoothed probability:

$$\begin{aligned} \sum_{t \in M \cap D} P(t|M) \left[ \log(P'(t|D)) - \log(\beta(t)) \right] = \\ \sum_{t \in M \cap D} P(t|M) \log \left( \frac{P'(t|D)}{\beta(t)} \right) = \\ \sum_{t \in M \cap D} P(t|M) \log \left[ \frac{\lambda P(t|D) + \beta(t)}{\beta(t)} \right] = \\ \sum_{t \in M \cap D} P(t|M) \log \left[ \frac{\lambda P(t|D)}{\beta(t)} + 1 \right] \end{aligned}$$

We now have raw probabilities in all terms. If we substitute back into Equation 4 we arrive at the following:

$$\begin{aligned} H(M \| D) = & \sum_{t \in M} P(t|M) \log(\beta(t)) + \\ & \sum_{t \in M \cap D} P(t|M) \log \left[ \frac{\lambda P(t|D)}{\beta(t)} + 1 \right] \end{aligned} \quad (8)$$

This formulation has two crucial advantages over Equation 4: the second sum is now over the *intersection* of $M$ and $D$, and all of the probabilities are raw probabilities. The cross-product of postings will in fact recover all of the terms in the second sum for every $M$,$D$ pair. In order to handle the first sum term from Equation 8, we can make a simple modification to the original postings list algorithm: we generate this sum while reading the document stream, and at the end of a document, we emit a tuple $\langle \emptyset, \sum_{t \in M} P(t|M) \log(\beta(t)) \rangle$, which now represents the background cross-entropy score of all documents with respect to $M$. During later processing, we need to take care treat the $\emptyset$ "term" differently than the other terms, however all we need to do downstream is carry the tuple forward to the final sums for all documents. This whole process only produces an extra $|C|$ increments to shuffle. Considering that the number of increments generated is

typically thousands of times more than $|C|$, the additional tuples are negligible.

## 2.3 Final fastRM scores for retrieval

So far we have discussed how to use the matrix $A$ to calculate the relevance model cross-entropy, $H(M\|D)$, quickly. Several researchers have shown that RMs can be improved if the original query's likelihood is interpolated into the score [1, 9]. We will use that variation, often called RM3.

Given a calculated matrix $A$, we use the following process to calculate final scores:

1. For query $Q$, perform retrieval using language modeling, producing ranked list $\hat{R}$.
2. Select the top *fbDocs* documents to form the feedback set $\mathcal{M} = \{M_1, M_2, \ldots\}$
3. Calculate the posterior scores, $P(M_i|q_1 \ldots q_i)$ (see Eq. 3). We set $P(D|q_1 \ldots q_i) = 0$ for all $D \notin \mathcal{M}$.
4. For each document $M_i \in \mathcal{M}$:
   (a) Retrieve $M_i$'s row in the matrix, $A$.
   (b) Scan the row, accumulating the score (posterior times the value in $A$) for any document encountered.
5. Interpolate the original ranked list score with the calculated cross-entropy score.
6. Return the top scoring documents from the resulting list as the final ranked list.

# 3. EVALUATING FASTRM

We implemented the offline calculation using Hadoop MapReduce v0.20.1, and processing was performed on Yahoo! Inc.'s M45 cluster[1]. For retrieval, we modified a copy of Indri 2.10 [27] to support merging the previously calculated scores into the ranked list.

We conduct our experiments over 3 collections, shown in Table 2. We use the built-in Krovetz stemmer and the INQUERY 418-word stopword list during indexing. We use topics 51-200 from the early ad-hoc tracks of TREC[2] as the query set for AP89. We use only topics 151-200 from the same topic set for experiments with the Wall-Street Journal (WSJ) collection. The Robust05 collection consists of the AQUAINT document collection, with the topic set taken from the TREC Robust 2005 track. This set consists of 50 topics. We use only the title text of each topic. For all PRF experiments, we set the number of feedback documents ($fbDocs$) to 10, and the number of feedback terms ($fbTerms$) to 100. The $fbTerms$ parameter does not affect the fastRM implementation.

We report Mean Average Precision (MAP) and Normalized Discounted Cumulative Gain (NDCG) in order to gauge retrieval performance between different methods. We use `trec_eval` v9.0 for this purpose. The Language Model and the Relevance Model act as the baseline methods for our experiments, the former serving as the efficiency baseline, and the latter as the effectiveness baseline. In short, we want Language Model speed with Relevance Model accuracy. We use the paired sample randomization $t$ test as described by Smucker and Carterette [25] for significance testing. We use 10 million samples for each significance test unless otherwise noted.

[1]http://research.yahoo.com/node/1884
[2]http://trec.nist.gov

| Collection | docs ($10^3$'s) | terms ($10^6$'s) | unique ($10^3$'s) | avg \|D\| (1's) | $\|A\|$ (Gb's) |
|---|---|---|---|---|---|
| AP89 | 84.6 | 42.1 | 211.5 | 497.8 | 54.9 |
| WSJ | 173.2 | 81.7 | 243.5 | 471.6 | 228.8† |
| Robust05 | 103.3 | 484.2 | 892.2 | 468.6 | 8105.9† |

**Table 2: Statistics about collections used. † indicates an estimated value.**

| Run | NDCG | MAP | Ret. Time (msecs) | Build (mins) | Size (GB's) |
|---|---|---|---|---|---|
| LM | 0.3650 | 0.1972 | 13.1 | N/A | N/A |
| RM3 | 0.4115† | 0.2377† | 2506.9 | N/A | N/A |
| fastRM3 | 0.4006♦ | 0.2224♦ | 800.8 | 329.9 | 54.9 |

**Table 3: Comparing the Language Model, RM3, and fastRM models on AP89. 'Build' indicates the time to construct the matrix $A$. 'Size' indicates space consumed by the compressed matrix. ♦ indicates statistical significance over LM ($p < 0.01$). † indicates statistical significance over both LM and fastRM3 ($p < 0.01$).**

Table 3 shows retrieval results over the AP89 collection for the two baseline methods and the fully pre-computed fastRM. The fastRM shows a clear improvement over the Language Model in terms of retrieval effectiveness, and—even on just this small collection—a considerable reduction in retrieval time compared to the Relevance Model.

# 4. MAKING FASTRM FASTER

Although we have shown that fastRM can be slightly slower than LM while providing much of the gain of RM, there are three issues that appear as the collection size grows:

1. The time needed to calculate the matrix is reasonable for the AP89 collection, but will grow unacceptably with more documents – even using cloud-based approaches such as MapReduce.
2. Even when the full matrix can be calculated, storing it can become a challenge since it grows quadratically with the collection size. Table 2 shows sizes, estimated or known, of the fully computed matrix for each collection.[3]
3. Finally, the growing storage requirements in turn increase the time to scan matrix rows during retrieval. Eventually, scanning a single row will take long enough to nullify any speed advantage obtained via precomputation.

In the rest of this section we will present methods for addressing the issues above. In Section 4.2 we explore methods that reduce the size of the stored matrix, addressing the second and third points above. In particular, we demonstrate

[3]For the WSJ and Robust05 collections we only calculated the portions of the matrix used in our experiments and estimated the final size based on the partial matrix.

that large portions of the matrix can be discarded without sacrificing the gains in effectiveness. Then, in Section 4.3, we address the first point by discussing methods that attempt to generate a similarly sparse matrix by omitting the calculations of many entries from the start. First we discuss measuring the quality of matrix approximation.

## 4.1 Measuring approximation

In this section we will be approximating the original matrix $A$. That is, we will remove entries, calculate them less accurately, and so on, resulting in a matrix $\hat{A}$. Although the ultimate impact of using $\hat{A}$ is measured by retrieval effectiveness, we will also compare the approximated and true matrices directly as described here.

The ranked list evaluation measures inform us how an approximation $\hat{A}$ impacts retrieval. However, when comparing a fully calculated row from $A$ and an approximate construction from $\hat{A}$, we would like to know how much of the original matrix is recovered by the approximation. Intuitively, the documents in a row with higher cross-entropy relative to the given $M$ should be considered more important than documents with a lower cross-entropy: in Equation 3 they contribute more to the score. Therefore we desire a method that assigns more importance to documents at higher ranks in a particular row. We would also like a measure that is bounded, since by definition the best performance we can hope for is recovering exactly the elements of the row we specify.

We adapt the NDCG measure [14] to fulfill this role. We call it NDCG$_{row}$ to disambiguate this measure from standard NDCG (which we report for retrieval evaluation). Let $A_i$ be row $i$ in a fully calculated matrix of cross-entropy values. Let $\hat{A}_i$ be the corresponding row $i$ in $\hat{A}$. All of the approximations used in this work compute the actual cross-entropy between two documents. The rank order is preserved, even if some documents do not appear in the approximate row, so if $|A_i|$ represents the number of non-empty entries in row $A_i$, then $|A_i| \geq |\hat{A}_i|$. Since standard NDCG is defined over two lists of the same length, we need to treat $\hat{A}_i$ as if it has the same number of entries as $A_i$. To do this we simply inject a 'non-relevant' entry into $\hat{A}_i$ in place of every document not recovered, to create a new list $\hat{A}'_i$. For example, if

$$A_i = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$$
$$\hat{A}_i = \{1, 6, 7, 8, 10\}$$

then

$$\hat{A}'_i = \{1, \times, \times, \times, \times, 6, 7, 8, \times, 10\}$$

Define $rel(id)$ to be 1 if document $id$ is not a $\times$ symbol, and 0 otherwise. We define $DCG@p$ for row $A_i$ as:

$$DCG@p(A_i) = \sum_{j=1}^{p} \frac{rel(A_i[j])}{\log_2(j+1)} \quad (9)$$

Now we can define NDCG$_{row}@p$ to be the ratio of the discounted cumulative gain (DCG) of $\hat{A}'_i$ over the DCG of $A_i$ at position $p$:

$$NDCG_{row}@p = \frac{DCG@p(\hat{A}'_i)}{DCG@p(A_i)} \quad (10)$$

Simply put, we use the non-relevant entries to make sure the actual entries in $\hat{A}_i$ are assigned the proper gain during computation. Using the example above, the measure's progression over several cutoffs is shown in Table 4.

| $p$ | 10 | 9 | 8 | 7 | 6 | 5 |
|---|---|---|---|---|---|---|
| NDCG$_{row}$@p | 0.552 | 0.518 | 0.562 | 0.520 | 0.465 | 0.390 |

**Table 4: NDCG$_{row}$ scores at different $p$ values.**

The drop in $NDCG_{row}$ makes sense — $\hat{A}_i$ recovers the documents at higher ranks, while missing the low rank documents. Therefore, as we decrease $p$ those gains are removed, decreasing the $NDCG_{row}$ score. Note that the earlier the gain occurs the more it increases the numerator. This illustrates the bias towards higher rankings that we desire.

## 4.2 Reducing Storage Requirements

We first examine whether we can reduce the size of the matrix that must be stored.

### 4.2.1 Column Dropping

In their original work, Lavrenko and Allan showed that one only needs to retain some number of the highest scores in each row, and performance is not notably impacted. We conduct similar experiments here. We define $\rho$ to be the number of values retained for each row. If we consider the rows to be left-to-right sorted in score-descending order, for $\rho = 1000$, that means we retain the 1000 leftmost columns in the matrix. For the document scores dropped, we simply use the background score for the row in question—i.e., $H(M\|D)$ when $M \cap D$ is empty.

Intuitively, we can think of this dropping as increasing the distinction between the $\rho$ more similar documents and the remaining less similar documents for a given model $M$. Figure 3 shows a comparison of the different values of $\rho$ against the full calculation from Section 3. Similar to Lavrenko and Allan, retaining fewer scores has negligible, and in some cases, a slightly positive effect. More notably, since we stop scanning each row early, the retrieval time has dropped dramatically, almost to the LM time: Table 5 shows the specifics. We cannot only get away with dropping some columns from the matrix; it appears that we *should*. Note that we cannot push the reduction down too far; retrieval performance drops markedly for a $\rho$ value of 10. Figure 4 shows different values of $\rho$ across the different collections. The upper bound of effectiveness seems to hover steadily around the $\rho = 100$ position, suggesting the parameter value to be insensitive to values above 100.

### 4.2.2 Binning

Several previous works [26, 2] have shown that binning retrieval scores can simultaneously increase efficiency and reduce space requirements while not significantly impacting retrieval performance. We investigate the viability of using binning here to reduce the number of unique cross-entropy scores that need to be stored. If we can reduce the number of unique values needed, say to 256, we only need to store 256 explicit values, while the rest could be single-byte reference entries. We experiment using two binning techniques: $\epsilon$-based binning, and stepwise binning. We implement both methods as a function of the actual scores produced at retrieval time.

| $\rho$ | AP89 | WSJ | Robust05 |
|---|---|---|---|
| 0 | 13.11 | 31.92 | 66.13 |
| 100 | 19.22 | 37.92 | 146.92 |
| 1000 | 25.66 | 44.84 | 157.76 |
| 10000 | 110.50 | 138.52 | 262.60 |
| $\infty$ | 800.80 | 2125.29 | 18131.89 |
| RM | 2506.90 | 5124.07 | 28494.73 |

**Table 5: Query-processing times for different values of $\rho$. Time is in milliseconds. $\rho = 0$ is the Language Model run. $\rho = \infty$ is using the entirety of the provided matrix.**

**$\epsilon$-based**. For a given retrieval run, we define the variable $\epsilon$ to be the amount two scores must differ by in order to create a new bin. Given two row-wise adjacent entries in a matrix, $A_{i,j}$ and $A_{i,j+1}$, if $|score(A_{i,j}) - score(A_{i,j+1})| < \epsilon$, then $A_{i,j+1}$ is placed in the same bin as $A_{i,j}$ and uses the same score that $A_{i,j}$ is using (which itself may be a surrogate score). Otherwise we create a new bin, using $score(A_{i,j+1})$ as the score for that bin. Intuitively, as $\epsilon \to 0, \#$ bins $\to |C|$. Inversely, as $\epsilon \to \infty, \#$ bins $\to 1$.

**Stepwise binning.** The step function uses two values, the binsize ($b$) and the number of bins ($n$). For example, a bin size of 100 with 10 bins means that the first 1000 documents are placed into 10 bins, where the first bin contains the 100 most highly scored documents, the second bin contains the 100 next documents, and so on. For each bin, the highest score in a that bin is used for all of the documents contained in the bin. If the number of documents in the row is greater than the number specified by $b \times n$, all of the remaining documents are placed in the last (i.e. rightmost) bin.

Figure 5 shows results for several values of $\epsilon$. Although the best run for each collection is better than corresponding worst run by a statistically significant amount, the change is not particularly dramatic across the runs. More importantly, none of the $\epsilon$-binned runs perform worse than the run where binning was not used (leftmost column for each collection). This suggests that we could reorganize our matrix to only store the values that start a bin, and delete the other scores in each bin.

Figure 6 shows the results of similar experiments for the stepwise binning function. Stepwise binning is noticeably more sensitive to value changes than $\epsilon$-binning. Although none of the runs precipitously drop off in performance, it seems that $\epsilon$-binning will require less tuning to find an acceptable value.

## 4.3 Reducing Offline Computation Time

The results in Section 4.2 indicate that several ways exist to reduce the amount of data we need to store while retaining the information we need to improve retrieval performance. The binning techniques appear effective, but they only reduce the amount of data we need to store after computation is complete; we still keep track of all scores. The more intriguing outcome is the result of dropping columns from the matrix completely. We can drop columns to reduce the amount of space needed, but we would *rather* refrain from calculating the entire matrix in the first place. However, this presents us with a chicken-and-egg problem: If we do not have the pairwise similarity scores, how do we know which document pairs need to have their scores calculated?

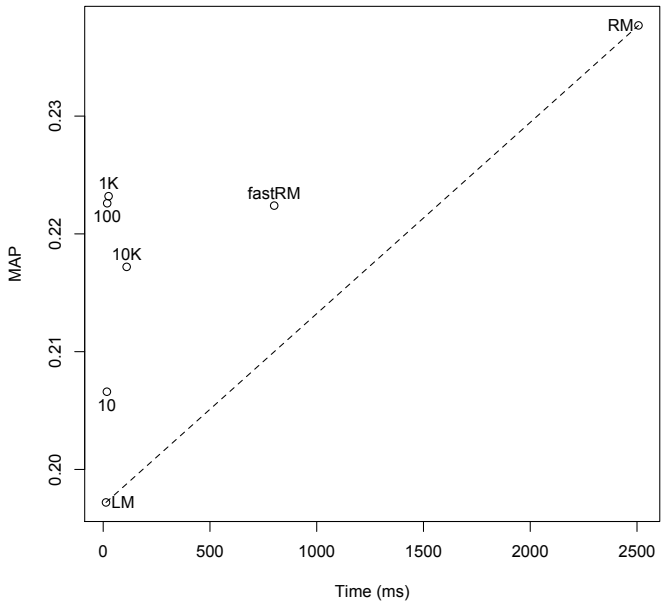


**Figure 3: A comparison of MAP scores to query-processing time for the AP89 collection. The label indicates how many entries are retained per row (i.e., $\rho$). Other collections show similar characteristics as $\rho$ varies.**

To address this issue, we explore methods that project our document space to a lower dimensional space while attempting to preserve the locality of the documents. We cluster documents into groups and only calculate matrix entries for pairs of documents in the same cluster. For example, if we were to break the collection into two evenly sized clusters, we would skip computing half of the matrix entries.

### 4.3.1 Existing approaches

To achieve our goal we first turn to several algorithms that have been used for this kind of clustering in other domains: Locality Sensitive Hashing (LSH), random forests of *kd*-trees, and the Random Projection tree (RP-tree). We briefly introduce each method here.

Locality sensitive hashing [13] techniques have been frequently used to quickly estimate which documents are similar via bucket-hashing, thereby avoiding unnecessary comparisons between documents that are unlikely to similar to each other. Broder et al. provided the first notable application of LSH to web collections [6]. Additionally, Indyk has described a simple construction method for the necessary hash functions for LSH [12]. We use this construction here. We generate fingerprints of each document, where the length of each fingerprint and the number of fingerprints from each document are parameters. We consider two documents as a candidate pair if they generate fingerprints that hash to the same bucket.

*kd*-trees [5, 11] are partitions of the space of data in order to reduce probe time when given a query point in the space.

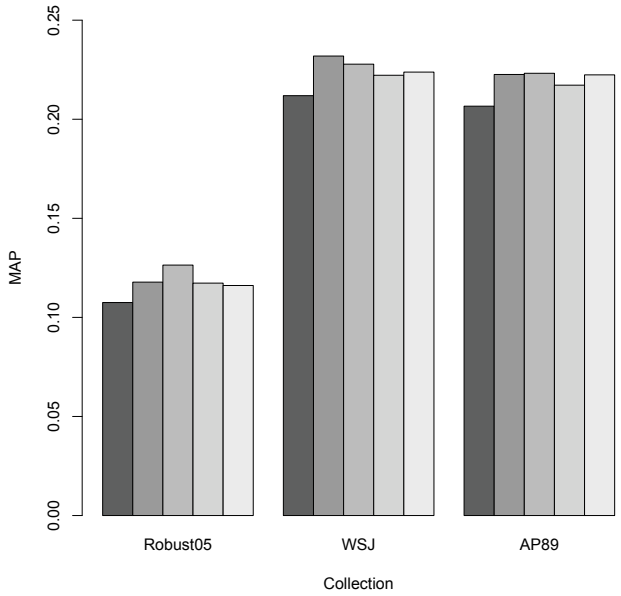


**Figure 4: MAP scores for various values of $\rho$. For each collection, from left to right, the values used are: 10, 100, 1000, 10000, and the entire matrix. The differences in MAP scores between $\rho = 10$ and the other values are statistically significant ($p < 0.05$) with respect to each collection.**

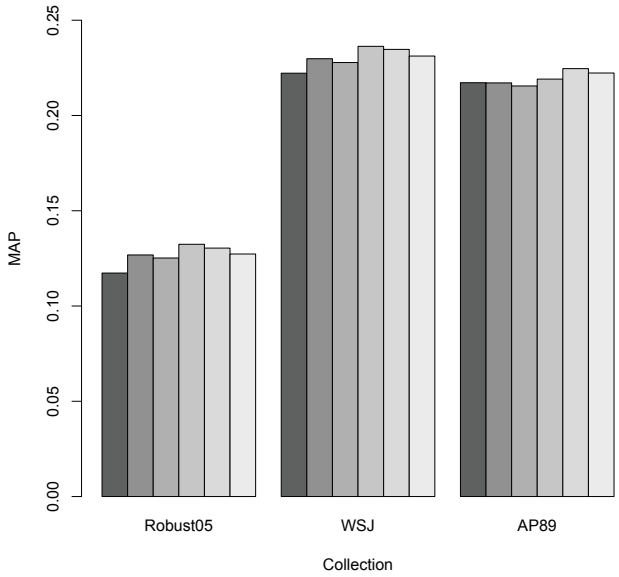


**Figure 5: Results for binning using $\epsilon$-binning. The leftmost value is the run without binning. The remaining value are, left to right: 0.1, 0.5, 1, 1.5, and 1.75. The best run in each collection is significantly ($p < 0.01$) better than the bottom 2 runs. Note that the best setting for $\epsilon$ is different for AP89 (1.5 vs. 1.0 for the other collections).**

Optimally the partitioning results in $\log n$ search time. A single *kd*-tree is known to operate poorly when the number of dimensions is high; search times usually approach $O(n)$ under such conditions. Recently Silpa-Anan and Hartley utilized multiple randomized *kd*-trees [24] to maintain efficient search over high-dimensional representations. Our implementation of *kd*-trees is similar in nature to those presented by Silpa-Anan and Hartley. Instead of partitioning the space for efficient querying, we treat the partitions as clusters for choosing pairs to evaluate. We generate pairs by taking the cross-product of a cluster vector with itself.

The Random Projection tree, or RP-tree, is a recent modification to the *kd*-tree [8]. The major improvement comes from the ability of an RP-tree to adapt to the intrinsic dimensionality of the data, whereas a *kd*-tree simply bisects a chosen dimension along the median value. Our hope here is that using an RP-tree will exploit structure in the term distribution space to better cluster similar documents together, generating a higher quality set of document pairs to calculate. We adopt a streaming version of the RP-tree, as described by Dasgupta and Freund[7].

**Results.** Table 6 shows the results of the methods described above, compared to the baseline methods and using the full matrix. We show the best runs of each method in the table, after experimenting with several parameter settings for each method.

None of the methods performed as we had hoped. The LSH build time is comparable to the full fastRM, but the process resulted in a matrix too small to even hold enough data to be effective. We had to filter out a large number of false positive matches to combat introducing noise into the final matrix. The majority of the generated fingerprint pairs were discarded as false positives. We also had difficulty choosing parameters for the algorithm. Targeting a similarity threshold of 0.2, we had to keep the fingerprint size as small as possible ($fsize = 1$) to provide any chance for matches in the collection, while using a large number of fingerprints ($fnum = 300$) to provide a decent chance for documents to intersect. The AP89 collection may be smaller than the intended domain of the LSH algorithm, but our difficulties in reducing false positive matches (i.e., low-similarity matches) indicate to us that LSH is not suitable here.

Both the random forest of *kd*-trees and the RP-tree exe-

| Run | NDCG | MAP | Build (mins) | Size (GBs) | $\text{NDCG}_{row}$ @1$K$ |
|---|---|---|---|---|---|
| LM | 0.3650 | 0.1972 | N/A | N/A | N/A |
| RM3 | 0.4115 | 0.2377† | N/A | N/A | N/A |
| FRM3 | 0.4006 | 0.2224† | 329.9 | 54.9 | 1.0 |
| LSH | 0.3659 | 0.1984 | 573.6 | 0.026 | 0.439 |
| *kd*-trees | 0.3496 | 0.1938 | 58.2 | 10.7 | 0.211 |
| RP-trees | 0.3728 | 0.2029† | 87.0 | 21.8 | 0.412 |

**Table 6: Retrieval results using the AP89 collection, topics 51-200. † indicates statistical significance of $p < 0.05$ over LM retrieval.**

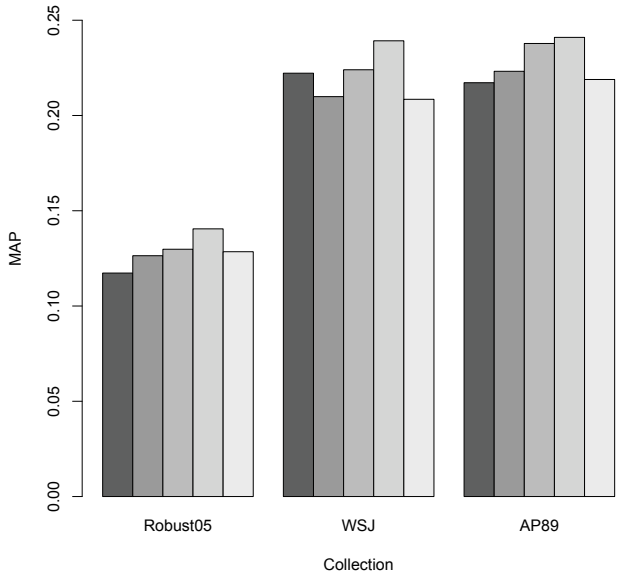


**Figure 6: Results for stepwise binning for $\rho = 1000$. Step values used: 1-1000, 10-100, 100-10, and 1000-1. Note that the 1-1000 (leftmost) run is equivalent to not using binning at all. The best 100-10 run for both WSJ and Robust05 are significantly ($p < 0.001$) better than the other runs.**

cuted in much less time and produced a smaller matrix than the original fastRM, however the retrieval results indicate that both matrices are of unacceptably low quality. All of the retrieval measures decreased slightly for *kd*-trees. Note also that the NDCG$_{row}$@1K score is very low, indicating that for most rows, few of the desired top 1K documents were recovered. We believe this due to a poor fit between feature selection for the algorithm and the features that effectively express the locality of the documents. *kd*-trees rely on globally selected features that are then used to partition the space. In the parlance of IR, the feature selection is tantamount to selecting terms based solely on their *idf* score, while disregarding the $tf$ score.

The RP-tree slightly increased some of the retrieval measures. However considering the size of the matrix, the effect should be much more pronounced. The NDCG$_{row}$@1K measure is better than the *kd*-tree, but it is still much lower than expected.

### 4.3.2 Using Domain Knowledge: Highpass

We believe the randomness introduced by the LSH and RP-tree algorithms make the algorithms theoretically appealing, but less efficient than if they simply used domain knowledge. The random forest of *kd*-trees fails to utilize local information necessary to effectively cluster. Therefore, we need an algorithm that projects each document into some lower dimensional space while preserving locality, but avoids involving randomness.

In order to preserve document locality as effectively as possible, the most important terms in each document must be part of the projection. When processing a document, we order all of the unique terms by some impact function $\mathcal{I}$. We set some threshold $\tau$, and only emit the first $\tau$ unique terms from each impact-ordered document. In this manner we view the projection process as a high-pass filter. We then process the resulting posting lists as described in Section 2.2. However instead of scoring the documents, we only form a list of document pairs based on the intersections within the term posting lists. Using this list we then generate the exact cross-entropy scores to form the approximate matrix. We set 'tf-idf' as the $\mathcal{I}$ function, while varying $\tau$.

Figure 7 shows results for different settings of $\tau$ compared to the Language Model (LM) and best fastRM (full) runs for the respective collections. All values have been scaled to reflect the increase from the lowest to the highest value for that particular axis and collection. Therefore a value of 0.5 indicates the actual value to be $low + 0.5(high - low)$. Figure 7 shows that for $\tau = 10$ and $\tau = 20$, the the build times are relatively low, but the relative increase in MAP is substantial. The runs where $\tau = 100$ took longer to construct than the original full calculation were due to how the cross-entropy scores are calculated from the resulting list of document pairs. It demonstrates that although high $\tau$ value does increase the build time as expected, it does not bring increased accuracy.

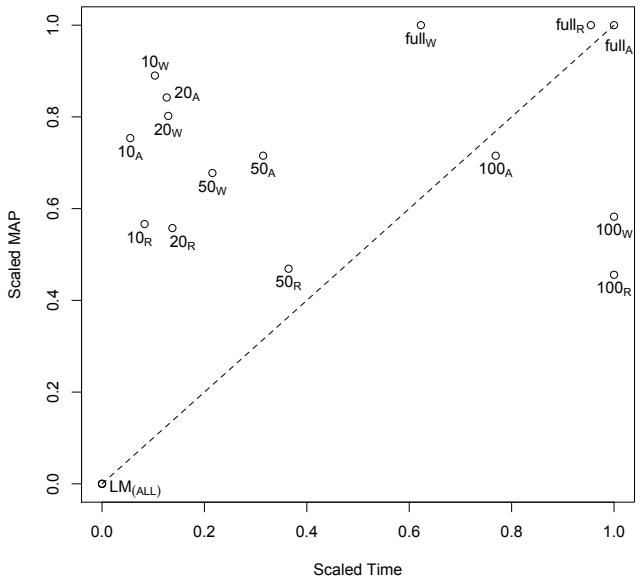


**Figure 7: Graph comparing increase in retrieval performance versus time to build the matrix used, over all collections. 'A','W', and 'R' indicate AP89, WSJ, and Robust05 respectively. Neither column-dropping nor binning were performed on the approximate matrices used. Labes are the value of $\tau$.**

Table 7 provides some insight into why the Highpass algorithm performs well with so few terms projected. Even with $\tau = 10$, the NDCG$_{row}$@1K is above 0.5 for all 3 collections, better than all of the ealier methods. One can also see the

| Collection | Value of $\tau$ 10 | 20 | 50 | 100 |
|---|---|---|---|---|
| AP89 | 0.524 | 0.727 | 0.921 | 0.984 |
| WSJ | 0.567 | 0.786 | 0.948 | 0.984 |
| Robust05 | 0.718 | 0.887 | 0.977 | 0.990 |

**Table 7: $NDCG_{row}$@1K of Highpass for the different values of $\tau$, across all collections.**

diminishing returns in increasing the value of $\tau$. This suggests that while the $NDCG_{row}$ increases as expected, more low-rank documents are also being pulled into the lists. This explains the phenomenon of the $\tau = 100$ runs actually performing worse than the runs with fewer terms used. This begs the question of how much the performance is tied to the number of terms projected, as opposed to just picking the *right* terms to project.

# 5. RELATED WORK

We briefly sketch related work a few connected areas.

**Pseudo-Relevance Feedback.** The literature on PRF is too large to cite in its entirety here, but we reference previous influential works that provided much of the basis for this work. The classic expansion technique for vector space models is the Rocchio algorithm [23], which uses the original ranked list to reweight the query vector. The theoretical foundation of this work comes from the Relevance Model [17], developed by Lavrenko and Croft in 2001. At a similar time, Zhai and Lafferty produced a method called model-based feedback [28]. Both the relevance model and model-based feedback operate as pseudo-relevance feedback mechanisms for Language Models [21]. Recently, Metzler and Croft developed Latent Concept Expansion [20], an expansion mechanism for the Markov Random Field [19] (MRF). Metzler and Croft contrast the Language Model/Relevance Model pairing in parallel to the MRF/LCE pairing. Juxtaposing the models in this way shows that we cannot directly compare to LCE, however it seems like a natural progression for work of this kind.

**Distributed processing & Efficiency.** Some recent work toward performing pairwise document comparisons in a distributed environment comes from Elsayed et al. [10], who showed a simple MapReduce algorithm to compute the inner product scores over a given collection. In 2009, Lin continued this line of work, presenting two new algorithms for computing inner product scores in a MapReduce environment [18]. Anh and Moffat have investigated using impact-order of terms to increase retrieval efficiency [2, 3]. While our Highpass algorithm is similar in spirit, the application is different. Additionally, Anh and Moffat conduct their experiments using a Vector Space Model, whereas we operate in a probabilistic retrieval setting.

**All-Pairs Computation.** Bayardo et al. also presented a modification to the all-pairs calculation problem, exploiting various features of data sparsity to significantly reduce the time required to compute scores over all pairs [4]. Although their algorithm looks appropriate, it depends on the pairwise similarity function being a proper metric, which ours is not. More recently, Kang et al. presented PEGASUS, a new framework which uses Hadoop MapReduce as a substrate to process large-scale matrices [15]. PEGASUS appears to operate on dense matrix representations, whereas we do not generate a such a representation except as final output of the offline computation.

# 6. CONCLUSIONS AND FUTURE WORK

We have shown that we can reformulate the traditional Relevance Model to allow for much of the computation to occur offline. Using this modification we can significantly improve retrieval performance over an unexpanded query with a slight additive increase in query-processing time. We have also discovered that the offline computed scores can be binned with no discernible negative impact on retrieval performance. This evidence suggests that we may be able to store less than 1% of the unique scores in a matrix row while sacrificing nothing in retrieval effectiveness.

Additionally, we can save space by dropping low-quality columns from the matrix. More importantly, we can reduce the computational requirements of calculating the matrix by trying to accurately predict the high-quality document pairs and only producing values for those entries. We experimented with several methods to achieve this goal. Results indicate that at least one method, the Highpass algorithm, shows considerable promise. We believe this work demonstrates the potential of our approach towards improving efficiency of theoretically sound query expansion. Furthermore, this work represents a significant step in bridging the gap between applying statistical pseudo-relevance feedback in a lab setting versus using it in the real world.

Although we have made progress towards achieving our goal, we see several avenues for continuing work based on the results here. We believe that we can improve the ratio of $NDCG_{row}$ vs. build time for the approximate algorithms. We have yet to consider different term weighting functions for $\mathcal{I}$ to see if another function may improve coverage over the desired set for each row. The Highpass algorithm is a good initial method to generate an approximate matrix, but we intend to explore other algorithms that may result in a better $NDCG_{row}$ score using the same amount of build time.

Like the original Relevance Model, the fastRM's retrieval performance is linked to the performance of the initial retrieval. While we only experimented with the Language Model here for the sake of speed comparisons, we are interested how using a more effective ranking function for the initial retrieval impacts the final performance of the fastRM.

Multiple variables require tuning in the system. While we have learned some of their characteristics here, much remains to be learned concerning their behavior and the relationships between them. We would like to develop some way to auto-tune these parameters based on collection statistics.

Ultimately, we would like some function that translates the amount of information loss between the "truth" matrix and the approximated matrix and the resulting performance. In short, given approximated matrix $A$ and collection $C$ we would like some function $f(A, C)$ that provides a measure of quality of $A$, so we can have some confidence that using $A$ will in fact provide some benefit to retrieval without *actually* testing the matrix against a set of judged queries, which may not always be available.

# 7. ACKNOWLEDGMENTS

This work was supported in part by the Center for Intelligent Information Retrieval, in part by NSF IIS-0910884, and in part by NSF CLUE IIS-0844226. Any opinions, findings

and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of the sponsor.

## REFERENCES

[1] N. Abdul-Jaleel, J. Allan, W. B. Croft, F. Diaz, L. Larkey, X. Li, M. D. Smucker, , and C. Wade. Umass at trec 2004 — novelty and hard. In *Proceedings of the Thirteenth Text Retrieval Conference (TREC-13)*, Gaithersburg, MD, USA, 2004. NIST.

[2] V. N. Anh and A. Moffat. Simplified similarity scoring using term ranks. In *SIGIR '05: Proceedings of the 28th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 226–233, New York, NY, USA, 2005. ACM.

[3] V. N. Anh and A. Moffat. Pruned query evaluation using pre-computed impacts. In *SIGIR '06: Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 372–379, New York, NY, USA, 2006. ACM.

[4] R. J. Bayardo, Y. Ma, and R. Srikant. Scaling up all pairs similarity search. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*, pages 131–140, New York, NY, USA, 2007. ACM.

[5] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, 1975.

[6] A. Z. Broder, S. C. Glassman, M. S. Manasse, and G. Zweig. Syntactic clustering of the web. *Comput. Netw. ISDN Syst.*, 29(8-13):1157–1166, 1997.

[7] S. Dasgupta and Y. Freund. Random projection trees and low dimensional manifolds. Technical report, San Diego, CA, USA, 2007.

[8] S. Dasgupta and Y. Freund. Random projection trees and low dimensional manifolds. In *STOC '08: Proceedings of the 40th annual ACM symposium on Theory of computing*, pages 537–546, New York, NY, USA, 2008. ACM.

[9] F. Diaz and D. Metzler. Improving the estimation of relevance models using large external corpora. In *SIGIR '06: Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 154–161, New York, NY, USA, 2006. ACM.

[10] T. Elsayed, J. Lin, and D. W. Oard. Pairwise document similarity in large collections with mapreduce. In *HLT '08: Proceedings of the 46th Annual Meeting of the Association for Computational Linguistics on Human Language Technologies*, pages 265–268, Morristown, NJ, USA, 2008. Association for Computational Linguistics.

[11] J. Friedman, J. L. Bentley, and R. A. Finkel. An algroithm for finding best matches in logarithmic expected time. Technical report, Stanford University, Stanford, CA, USA, 1976.

[12] P. Indyk. A small approximately min-wise independent family of hash functions. In *SODA '99: Proceedings of the tenth annual ACM-SIAM symposium on Discrete algorithms*, pages 454–456, Philadelphia, PA, USA, 1999. Society for Industrial and Applied Mathematics.

[13] P. Indyk and R. Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *STOC '98: Proceedings of the thirtieth annual ACM symposium on Theory of computing*, pages 604–613, New York, NY, USA, 1998. ACM.

[14] K. Järvelin and J. Kekäläinen. Cumulated gain-based evaluation of ir techniques. *ACM Trans. Inf. Syst.*, 20(4):422–446, 2002.

[15] U. Kang, C. E. Tsourakakis, and C. Faloutsos. Pegasus: A peta-scale graph mining system. *Data Mining, IEEE International Conference on*, 0:229–238, 2009.

[16] V. Lavrenko and J. Allan. Real-time query expansion in relevance models. IR 473, University of Massachusetts Amherst, University of Massachusetts Amherst, 2006.

[17] V. Lavrenko and W. B. Croft. Relevance based language models. In *SIGIR '01: Proceedings of the 24th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 120–127, New York, NY, USA, 2001. ACM.

[18] J. Lin. Brute force and indexed approaches to pairwise document similarity comparisons with mapreduce. In *SIGIR '09: Proceedings of the 32nd international ACM SIGIR conference on Research and development in information retrieval*, pages 155–162, New York, NY, USA, 2009. ACM.

[19] D. Metzler and W. B. Croft. A markov random field model for term dependencies. In *SIGIR '05: Proceedings of the 28th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 472–479, New York, NY, USA, 2005. ACM.

[20] D. Metzler and W. B. Croft. Latent concept expansion using markov random fields. In *SIGIR '07: Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 311–318, New York, NY, USA, 2007. ACM.

[21] J. M. Ponte and W. B. Croft. A language modeling approach to information retrieval. In *SIGIR '98: Proceedings of the 21st annual international ACM SIGIR conference on Research and development in information retrieval*, pages 275–281, New York, NY, USA, 1998. ACM.

[22] P. Ratanaworabhan, J. Ke, and M. Burtscher. Fast lossless compression of scientific floating-point data. In *DCC '06: Proceedings of the Data Compression Conference*, pages 133–142, Washington, DC, USA, 2006. IEEE Computer Society.

[23] J. J. ROCCHIO. Relevance feedback in information retrieval. *SMART Retrieval System Experimens in Automatic Document Processing*, 1971.

[24] C. Silpa-Anan and R. Hartley. Optimised kd-trees for fast image descriptor matching. In *Computer Vision and Pattern Recognition, 2008. CVPR 2008. IEEE Conference on*, pages 1–8, June 2008.

[25] M. D. Smucker, J. Allan, and B. Carterette. A comparison of statistical significance tests for information retrieval evaluation. In *CIKM '07: Proceedings of the sixteenth ACM conference on*

*Conference on information and knowledge management*, pages 623–632, New York, NY, USA, 2007. ACM.

[26] T. Strohman. *Efficient Processing of Complex Features for Information Retrieval*. PhD thesis, University of Massachusetts Amherst, University of Massachusetts Amherst, December 2007.

[27] T. Strohman, D. Metzler, H. Turtle, and W. B. Croft. Indri: a language-model based search engine for complex queries. *Proceedings of the International Conference on Intelligent Analysis*, 2005.

[28] C. Zhai and J. Lafferty. Model-based feedback in the language modeling approach to information retrieval. In *CIKM '01: Proceedings of the tenth international conference on Information and knowledge management*, pages 403–410, New York, NY, USA, 2001. ACM.