Approximating Matrix Multiplication for Pattern Recognition Tasks

Edith Cohen David D. Lewis
AT&T Labs – Research
Murray Hill, NJ 07974, USA.
{edith,lewis}@research.att.com

January 9, 1997

Abstract

[1] Many pattern recognition tasks, including estimation, classification, and the finding of similar objects, make use of linear models. The fundamental operation in such tasks is the computation of the dot product between a query vector and a large database of instance vectors. Often we are interested primarily in those instance vectors which have high dot products with the query. We present a random sampling based algorithm that enables us to identify, for any given query vector, those instance vectors which have large dot products, while avoiding explicit computation of all dot products. We provide experimental results that demonstrate considerable speedups for text retrieval tasks. Our approximate matrix multiplication algorithm is applicable to products of $k \geq 2$ matrices and is of independent interest. Our theoretical and experimental analysis demonstrates that in many scenarios, our method dominates standard matrix multiplication.

1. Introduction

We propose and analyze a random-sampling based algorithm that estimates all entries of a matrix product of $k \geq 2$ matrices. The estimates are unbiased. Furthermore, when the matrices are nonnegative, the distribution of the relative error depends only on the relative size of an entry (in its respective row or column), and converges for larger entries. For objectives such as identifying all high-valued entries in nonnegative matrix products, we exhibit polynomial asymptotic improvements over the time bounds of "exact" matrix multiplication. Our technique is well suited for pattern recognition applications. One such application is text retrieval, and we provide experimental results for several text retrieval tasks. For many tasks, both on sparse and dense matrices, our algorithm outperforms (asymptotically and in practice) matrix multiplication. A similar technique for estimating matrix-vector products ($k = 2$) was independently discovered by Clarkson [7, 8]. Clarkson is investigating the use of this method to speed up Gaussian elimination, the solving of linear systems of equations, and other mathematical programming applications.

In pattern recognition tasks, a database of instances to be processed (images, signals, documents,...) is commonly represented as a set of a vectors $\mathbf{x}_1, \ldots, \mathbf{x}_n$ of numeric feature values.

[1]A preliminary version of this paper appeared in the Proc. 8th ACM-SIAM Symposium on Discrete Algorithms, 1997.

Examples of feature values include the number of times a word occurs in a document, the coordinates of the center of mass of black pixels in an image, or the set of spectral coefficients output by a speech analyzer.

The desired processing of instances is often implemented using linear models [13, 20]. The knowledge required to carry out the processing is captured in a vector of numeric weights, $\mathbf{q} = (q_1, q_2, \ldots, q_d)$. Each instance is processed by taking the dot product of the weight vector $\mathbf{q}$ with each of the vectors $\mathbf{x}_1, \ldots, \mathbf{x}_n$ of feature values. The weights in $\mathbf{q}$ may be set by hand or by supervised learning from training data, in order that the products $\mathbf{q} \cdot \mathbf{x}_i$ satisfy some desired goal. In *classification* problems, the goal is that values of $\mathbf{q} \cdot \mathbf{x}_i$ are high when $\mathbf{x}_i$ is "related" to or is a member of a particular class represented by $\mathbf{q}$. In *regression* problems the goal is that $\mathbf{q} \cdot \mathbf{x}_i$ approximate some numeric value associated with instance $\mathbf{x}_i$.

In some cases, computing the proximity of one instance to another can be cast in linear form. For instance, if $\mathbf{q}$ and $\mathbf{x}$ are normalized to have Euclidean norm of 1.0 and are viewed as unit length vectors, then $\mathbf{q} \cdot \mathbf{x}$ is the cosine of the angle between them, a commonly used proximity measure. (In this case, the roles of instance vector and weight vector are interchangeable.)

These pattern recognition techniques are applied, for instance, in statistical information retrieval systems. Documents and queries are represented as vectors of terms or features. The user query can be treated as an example document, whose similarity to database documents must be found [22, ch. 4], or it can be used as evidence toward setting the parameters of a probabilistic classification function, which is then applied to database documents [10, 21]. In both cases, linear models are widely used and continue to be developed [19, 28].

We can view all the above techniques in terms of matrix multiplication. Applying a weight vector $\mathbf{q}$ to a database of instance vectors means computing the vector by matrix product $\mathbf{q}^T\mathbf{A}^T$, where $\mathbf{A}$ is a matrix whose rows $\{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$ are the instance vectors. The value of the ith entry of $\mathbf{q}^T\mathbf{A}^T$ corresponds to the dot product of $\mathbf{q}$ and $\mathbf{x}_i$. In proximity problems, we often need to compute dot products between all pairs of a set of instances. This is equivalent to computing the matrix product $\mathbf{A}\mathbf{A}^T$.

The scale of these matrix multiplications can be considerable. In text retrieval, for example, the number of instances (documents) can be 10^4 to 10^7 and the number of features (words or phrases) may be 10^5 or more. The resulting computation can be expensive, even when utilizing sparse matrix and indexing techniques [27]. Dense instances with hundreds of features are also common, for example, with factor analytic text representations (Sec. 5.1) or in image retrieval, and sparse matrix techniques are not applicable in these cases.

Our sampling algorithm identifies high-valued entries of nonnegative matrix products, without full computation of the product. The algorithm assigns *scores* to each entry of $\mathbf{q}^T\mathbf{A}^T$. In contrast to existing approximate scoring techniques (see [25]), the expected value of the scores is equal to the true value that would be obtained with the full computation. Furthermore, the variance of the score divided by the true value, is independent of the weight distribution of the instance vectors. It depends only on the (relative) value of the entry and decreases for higher-valued entries. Our algorithm is applicable to real-valued data but provides weaker worst-case bounds on the estimate quality. On the real-valued data sets we have tested, however, our algorithm achieves significant improvements over multiplication.

In many pattern recognition applications, the exact value of $\mathbf{q} \cdot \mathbf{x}$ is not critical, and our approximate dot products can be used instead of true ones. For instance, the feature values in vectors are often set by coarse heuristics, with the result that small differences in dot products between vectors have little consequence. Furthermore, many tasks inherently do not require exact values. In a classification task, we may only need to know whether $\mathbf{q} \cdot \mathbf{x}$ exceeds some threshold value for class membership. In an information retrieval system, we may want to find, for instance, the documents with the 10 highest scores and display them to the user, without needing exact knowledge of the score of any document. However, our algorithm is also applicable when exact top values are needed, as when the goal is to identify exactly the 10 largest dot products. In such tasks, the scores produced by our algorithm can be used to identify a small set of instances that is likely to contain the desired set of instances (e.g. those with large dot products). The exact dot product evaluation is carried out only for these selected instances.

The running time of our algorithm depends on the accuracy required for the task at hand and on the distribution of the scored values. Our algorithm is particularly suited for problems where a typical query $\mathbf{q}$ has a small number of relevant instances (high dot product values) and the bulk of instances have significantly smaller values. As a rough "asymptotic" comparison, the exact computation $\mathbf{m} = \mathbf{q}^T \mathbf{A}^T = (m_1, \ldots, m_n)$ of all dot products takes $O(dn)$ operations (for dense instances). Our proposed algorithm, for the task of isolating the highest dot product (possibly with some others which are very close in value), performs

$$O\left(\frac{\sum_{i=1}^n m_i}{\max_{i\in\{1,\ldots,n\}} m_i}\right)$$

(at most $O(n)$, but usually less) operations. Furthermore, each operation of the sampling algorithm is simpler: memory access and a counter increment versus a floating-point multiply-add. In experiments with large information retrieval datasets we demonstrate speedups of 5 to 15 fold without significantly compromising the accuracy of the results.

Matrix products estimation.

We state the time bounds and estimation quality of our matrix-product estimation. Let $\mathbf{A}_1, \ldots, \mathbf{A}_k$ ($k \geq 1$) be nonnegative matrices. These matrices can be viewed as a "fixed" database. Let $\mathbf{A}^{n_1 \times n_2} = \mathbf{A}_1 \cdots \mathbf{A}_k$ be their product. Our sampling algorithm preprocesses the matrices $\mathbf{A}_1, \ldots, \mathbf{A}_k$ in time linear in $\sum_{i=1}^k |\mathbf{A}_i|$ (where $|\mathbf{B}|$ denotes the *size*, meaning number of nonzero entries, of the matrix $\mathbf{B}$). Next, the algorithm inputs query vectors $\mathbf{q} \in Q$ (that can arrive online). Let

$$\mathbf{m} = \mathbf{q}^T \mathbf{A} = (m_1, \ldots, m_{n_2}) \ .$$

For each query $\mathbf{q}$, our algorithm outputs an *estimate vector* $\hat{\mathbf{m}} = (\hat{m}_1, \ldots, \hat{m}_{n_2})$. It also computes the sum $M = \sum_{i=1}^{n_2} m_i$ in linear time in $|\mathbf{q}|$, (using the columns-sums of $\mathbf{A}$ that are computed during preprocessing).

We give bounds on both the relative and additive error of our estimates. For relative error, we show that for any $1 \geq p > 0$ and $\epsilon > 0$, in time $O(|\mathbf{q}| + kp^{-1}\epsilon^{-2}\log n_2)$, the estimate $\hat{\mathbf{m}}$ is such that with confidence at least $1 - 1/\operatorname{poly}(n_2)$,

$$\forall i \in \{1, \ldots, n_2\} \text{ such that } \frac{m_i}{M} \geq p, \ \frac{|\hat{m}_i - m_i|}{m_i} \leq \epsilon \ ,$$

and moreover, (with the same confidence)

$$\forall i \in \{1, \ldots, n_2\}, \ \frac{\hat{m}_i}{M} < p(1 - 1.5\epsilon) \ \text{ implies } \frac{m_i}{M} < p \ .$$

In particular, all entries in a row larger than the mean of entries for the row (including, of course, the largest entry) are such that $\frac{m_i}{M} \geq \frac{1}{n_2}$. Therefore, all entries greater than the mean are identified and estimated to within relative error ϵ in $O(|\mathbf{q}| + kn_2\epsilon^{-2}\log n_2)$ time.

Our bound here compares favorably with the $O(|\mathbf{q}| + \sum_{i=1}^{k} |\mathbf{A}_i|)$ time per query of explicitly computing $\mathbf{m}$. Consider, for example, the product of k nonnegative dense matrices of dimension n (that is, $|\mathbf{A}_i| = n^2$ for all i, $|Q| = n$ and $|\mathbf{q}| = n$ for all $\mathbf{q} \in Q$.) Explicit computation requires $O(kn^3)$ operations (or $O(kn^{2.38})$ if fast matrix multiplication is used [9]). In contrast, estimating to within a small fraction all above-average entries in each row of the product requires $\tilde{O}(kn^2)$ time.

We also obtain uniform additive error bounds for *all* entries of $\mathbf{m}$: For any $\epsilon > 0$, in time $O(|\mathbf{q}| + kM^2\epsilon^{-2}\log n_2)$,

$$\text{with probability } 1 - 1/\operatorname{poly}(n_2),\ \forall i \in \{1, \ldots, n_2\},\ \ |\hat{m}_i - m_i| \leq \epsilon\ .$$

Real-valued matrix products.

Our basic sampling algorithm applies directly to nonnegative vectors and matrices. Nonnegativity is common but not universal in pattern recognition tasks. Negative values occur moderately often in query vectors, particularly when the query vector is actually a linear classifier or regression function. Negative values in instance vectors are less common, but occur in signal processing tasks and occasionally even in text processing (see Sec. 5.1). Our algorithm extends for real-valued matrices, but weaker bounds apply. We estimate real-valued matrix products by representing each product as the *difference* of two positive quantities, which we named, the *positive* and the *negative contributions* of the product. Each contribution can be estimated using a modification of our algorithm for nonnegative products.

Let $\mathbf{A}_1, \ldots, \mathbf{A}_k$ and the query $\mathbf{q}$ be real-valued. Let $\mathbf{A}$, $\mathbf{m} = \mathbf{q}^T\mathbf{A}$, $\hat{\mathbf{m}}$, and M be defined as in the nonnegative case above. As is the case for nonnegative data, the preprocessing step takes time linear in $\sum_{i=1}^{k} |\mathbf{A}_i|$. For $i = 1, \ldots, k$, let $\mathbf{A}_i^+$ be the matrix $\mathbf{A}_i$, where every entry is replaced by its absolute value. The vector $\mathbf{q}_+$ is defined similarly. Let

$$\mathbf{m}' = \mathbf{q}_+^T\mathbf{A}_1^+ \cdots \mathbf{A}_k^+ = (m'_1, \ldots, m'_{n_2})\ .$$

(The vector $\mathbf{m}'$ is the sum of the positive and negative contributions of the product $\mathbf{q}^T\mathbf{A}_1 \cdots \mathbf{A}_k$.) Let $M' = \sum_{i=1}^{n_2} m'_i$. In time $O(|\mathbf{q}| + kM'^2\epsilon^{-2}\log n_2)$,

$$\text{with probability } 1 - 1/\operatorname{poly}(n_2),\ \forall i \in \{1, \ldots, n_2\},\ \ |\hat{m}_i - m_i| \leq \epsilon\ .$$

The relative error of the estimate depends on the relative magnitudes of the positive and negative contributions, within their respective rows, and with respect to each other (bounds are provided in Section 3). Hence, it could be arbitrarily bad in the worst-case. Empirically, however, for real-valued data we had available, entries with large values typically had large positive contributions and small negative contributions (the contributions were inversely correlated), which allows for estimates with a small relative error.

Relation to Euclidean proximity problems.

A variety of tasks involve finding instances in close proximity to other instances. This may be a goal in itself, or may be a subtask in performing classification [11], regression [17], and other tasks.

Frequently seen proximity problems are *closest pair*, *nearest neighbors*, and *bichromatic nearest neighbors.* A variety of proximity measures can be used in such tasks. As mentioned earlier, the cosine correlation can be implemented directly as a dot product. Another common measure is the Euclidean distance, which is related to the dot product as follows:

$$||\mathbf{u}-\mathbf{v}||^2 = ||\mathbf{u}||^2 + ||\mathbf{v}||^2 - 2\mathbf{u}\cdot\mathbf{v} \ .$$

When vectors have Euclidean norm of 1.0, dot product determines Euclidean distance exactly. Simple reductions can handle vectors whose Euclidean norm is not 1.0. The caveat in applying our algorithm for Euclidean nearest neighbors is that a small relative error on estimating a dot-product translates to an additive error which is a fraction of the Euclidean norm. Additive errors on dot-product estimation, however, do result on similar additive errors on the corresponding Euclidean distances.

Euclidean proximity problems (likewise, computing all dot products) are trivially solvable in $O(dn^2)$ time (for dense data, for sparse data the time depends on the nonzero structure). Some known algorithms perform better when the dimension d can be treated as a constant (or is very small). Bentley *et al.*[3] showed better *expected* asymptotic performance for instances where the dimension is constant and the vectors are randomly sampled from certain distributions. Arya et al.[1] established better *worst-case* performance for constant dimension, by allowing approximation. They presented an $O(n \log n)$ time algorithm that computes $(1+\epsilon)$-nearest neighbors. (for any fixed constant ϵ and d.) The running time of these algorithms is near-linear in n, but exhibits exponential dependence on d. For high dimensions, even for the average or approximate problems, the $O(dn^2)$ bound is the best known (short of using fast matrix multiplication). Our algorithm exhibits significantly faster running times for realistic distributions. It is best suited for datasets where the ϵ-neighborhoods do not contain many points or when most of the near-neighbors are relevant and are far from the bulk of irrelevant points. It is easy to see that this is not the case for data generated uniformly at random. It is typical, however, for "real" datasets. For such distributions, the running time for all nearest neighbors could be $O(n^2 \log n)$.

In Section 2 we present and analyse our sampling algorithm for nonnegative matrix products. In Section 3 we generalize the sampling algorithm for real-valued matrices. In Section 4 we discuss the implementation. Our experimental results are presented in Section 5.

2. The sampling method for nonnegative products

Consider a product

$$\mathbf{P}^{n_1\times n_{k+1}} = \mathbf{A}_1^{n_1\times n_2}\cdots\mathbf{A}_k^{n_k\times n_{k+1}}$$

of $k \geq 2$ nonnegative matrices. We present our sampling algorithm for estimating the product $\mathbf{P}$. The algorithm inputs the matrices $\mathbf{A}_1,\ldots,\mathbf{A}_k$. In an online setting, the matrices $\mathbf{A}_2,\ldots,\mathbf{A}_k$ are fixed and provided as input at preprocessing time, and vectors $\mathbf{q}\in\mathcal{R}^{n_2}$ are presented online (this can be viewed as supplying the rows of the matrix $\mathbf{A}_1$ online). We first overview the structure (input, output, running time) of the algorithm.

- **Preprocessing stage:** Inputs the matrices $\mathbf{A}_2,\ldots,\mathbf{A}_k$ and creates appropriate data structures. Preprocessing requires linear time in the number of nonzero entries in $\mathbf{A}_2,\ldots,\mathbf{A}_k$.

- **Scoring the ith row of $\mathbf{P}$:** Inputs: $\mathbf{q} \equiv [\mathbf{A}_1]_{i\bullet}$ (the ith row of $\mathbf{A}_1$, where $i \in \{1, \ldots, n_1\}$), and number of samples S. (S can be obtained as input or determined online based on intermediate results).

  1. The row $[\mathbf{A}_1]_{i\bullet}$ is processed in linear time in the number of nonzeros in $[\mathbf{A}_1]_{i\bullet}$. As a byproduct, $M_i = \sum_{h=1}^{n_{k+1}} [\mathbf{P}]_{ih}$ is computed.
  2. Sample S times: Each sample is an index $j \in \{1, \ldots, n_{k+1}\}$, where j is selected with probability
  $$\frac{[\mathbf{P}]_{ij}}{\sum_{h=1}^{n_{k+1}} [\mathbf{P}]_{ih}} = \frac{m_j}{M_i} ,$$
  where
  $$\mathbf{m} \equiv [\mathbf{P}]_{i\bullet} = \mathbf{q}^T \mathbf{A}_2 \cdots \mathbf{A}_k = (m_1, \ldots, m_{n_{k+1}}) .$$
  The data structure created previously allows us to obtain samples with this property in $O(k)$ time. The sampling stage returns a *score vector* $(S_1, \ldots, S_{n_{k+1}})$, where S_j, the *score* of $j \in \{1, \ldots, n_{k+1}\}$, is the number of times j appears in the sample (We have $S = \sum_{j=1}^{n_{k+1}} S_j$). Since the expected value of S_j equals $m_j \frac{S}{M}$, the vector
  $$\frac{M_i}{S}(S_1, \ldots, S_{n_{k+1}})$$
  is an unbiased estimate of $[\mathbf{P}]_{i\bullet}$.

The sampling algorithm provides information about the weight distribution and values of $\mathbf{m}$ in a fraction of the time needed to compute $\mathbf{m}$ explicitly. A more elaborate analysis of the dependence of the estimate accuracy on the number of samples S is provided in Subsection 2.4.

2.1. The algorithm

The preprocessing stage constructs a $k + 1$-layer directed graph, where edges are directed from layer j to layer $j + 1$ and all edges are assigned transition probabilities. A sample for row i of A_1 is obtained by a random walk from the ith node in the first layer to the $(k + 1)$st layer. (The index of the last node reached constitutes the returned value.)

Graph representation of matrices and matrix-products.

A matrix $\mathbf{A}$ of dimension $n_1 \times n_2$ is represented by a weighted directed bipartite graph $G(\mathbf{A})$, with n_1 nodes $\{v_1, \ldots, v_{n_1}\}$ at the lower level and n_2 nodes $\{v'_1, \ldots, v'_{n_2}\}$ at the upper level. For $1 \le i \le n_1$ and $1 \le j \le n_2$, there is a directed edge from v_i to v'_j if and only if $[\mathbf{A}]_{ij} \ne 0$. The weight $w(e)$ of the edge $e = (v_i, v'_j)$ is $[\mathbf{A}]_{ij}$. It is easy to see that the outdegree of v_i (resp., indegree of v'_j) is the number of nonzeros in the ith row (resp., jth column) of $\mathbf{A}$. The sum of the weights of edges incident to v_i is the weight of the ith row (that is, the sum of the weights of all entries in the ith row).

The matrix product $\mathbf{P} = \mathbf{A}_1 \cdots \mathbf{A}_k$ (where $\mathbf{A}_i$ is of dimensions $n_i \times n_{i+1}$) is represented as a layered graph with $(k + 1)$ layers. For $1 \le i \le k + 1$, the ith layer has n_i nodes and the edges between the ith and the $(i + 1)$st layers are as in the bipartite graph $G(\mathbf{A}_i)$. For $1 \le i \le k + 1$, let $V_i = \{v^i_1, v^i_2, \ldots, v^i_{n_i}\}$ be the set of nodes at layer i.

Preprocessing stage.

The following algorithm assigns transition probabilities $p : E$ to the edges of the product graph such that random walks from node $v_1^i \in V_1$ to layer V_{k+1}, according to the edge probabilities, produce the desired samples. First, for every subproduct, $\mathbf{A}_i \cdots \mathbf{A}_k$ $(1 \le i \le k)$ and every $1 \le j \le n_i$, we compute the sum $W_j^i = \sum_{h=1}^{n_{k+1}} [\mathbf{A}_i \cdots \mathbf{A}_k]_{jh}$ (the weight of the jth row of $\mathbf{A}_i \cdots \mathbf{A}_k$), and associate it with the node v_j^i. We denote $W_j^i \equiv W(v_j^i)$ and $M_j \equiv W_j^1$.

Algorithm 2.1 [Compute transition probabilities]

- Compute W_j^i:
  1. Layer $k+1$ nodes: For $1 \le j \le n_{k+1}$, set $W_j^{k+1} \leftarrow 1$.
  2. Layer k nodes: For $1 \le j \le n_k$, set $W_j^k \leftarrow \sum_{h=1}^{n_{k+1}} [\mathbf{A}_k]_{ih}$ (the sum of the weights of outgoing edges from v_j^k to V^{k+1}).
  3. For $i = k-1$ down to 1: (ith layer nodes)
     For every node v_j^i in layer i, set W_j^i to the sum of the products $w(v_j^i, u)W(u)$, for all $u \in V^{i+1}$ such that (v_j^i, u) is an edge.
- Assign probabilities:
  For every pair of nodes $\{v_j^i, v_\ell^{i+1}\}$ such that the edge $e = (v_j^i, v_\ell^{i+1})$ exists, set

$$p(e) \equiv p_{j\ell}^i = \frac{w(e)W_\ell^{i+1}}{W_j^i} \ .$$

Scoring.

To sample S times from the ith row of $\mathbf{P}$: Trace S random walks from v_i^1 to layer $k+1$. For $1 \le j \le n_{k+1}$, let S_j, the *score* of j, be the number of random walks that terminated in v_j^{k+1}. The quantity $M_i \frac{S_j}{S}$ has expected value $[\mathbf{P}]_{ij}$.

2.2. Example

Consider the 3×3 matrices $\mathbf{A}$ and $\mathbf{B}$ and their product $\mathbf{AB}$:

$$\mathbf{A} = \begin{array}{|c|c|c|}\hline 4 & 2 & 0 \\ \hline 1 & 2 & 5 \\ \hline 0 & 3 & 6 \\ \hline\end{array} \quad \mathbf{B} = \begin{array}{|c|c|c|}\hline 2 & 3 & 0 \\ \hline 5 & 3 & 2 \\ \hline 0 & 1 & 4 \\ \hline\end{array} \quad \mathbf{AB} = \begin{array}{|c|c|c|}\hline 18 & 18 & 4 \\ \hline 12 & 14 & 24 \\ \hline 15 & 15 & 30 \\ \hline\end{array}$$

Figure 1 shows the graph representation of the product $\mathbf{AB}$. On the left, edges are labeled by their weight. On the right, nodes are labeled by the values W_i^1 and W_i^2 $(i = 1, \ldots, 3)$ and edges are labeled by the transition probabilities $p : E$.

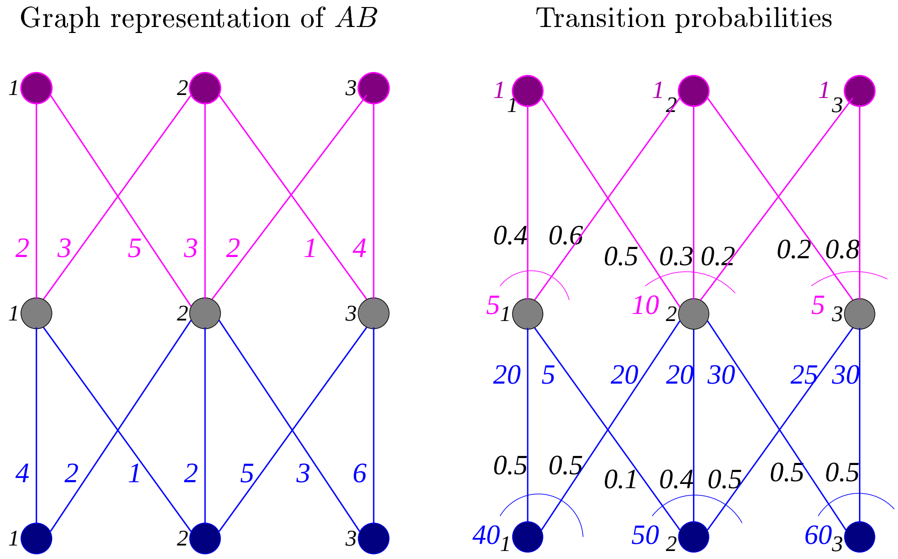

Figure 1: On the left we show the graph representation of the example matrix product. On the right we show the resulting node weights (i.e. row weights for subproducts), the products of these node weights with incident edge weights, and the resulting probability distributions on edges.

2.3. Correctness and running time

The preprocessing stage involves constructing the graph and computing transition probabilities (Algorithm 2.1), and it is easy to see that it can be implemented in $O(\sum_{i=1}^{k}|A_i|)$ time. Each sample amounts to a random walk of length k. We assume, for simplicity, that each step of the random walk takes $O(1)$ time units. We note, however, that the actual behaviour depends on the computational model and the implementation. In the bit-model of computation, each step of the random walk requires time proportional to the entropy of the distribution at the current node. In the common unit-cost model, each step requires at most $\log \deg(v) = O(\log n)$ time. In our implementation (see Section 4) we use *presampling*, that is, perform the sampling in the preprocessing stage and produce and store samples ahead of time. These samples are retrieved later for query evaluations. This make the actual cost *per sample* only $O(1)$ (as opposed to $O(k)$ without presampling).

Correctness is established by the propositions below.

Proposition 2.2 *For $i = 1, \ldots, k$ and $1 \le j \le n_i$, $W_j^i = \sum_{h=1}^{n_{k+1}} [\mathbf{A}_i \cdots \mathbf{A}_k]_{jh}$.*

Proof: By downward induction on $i = k, k-1, \ldots, 1$. Immediate for $i = k$. Assume the proposition holds for $i > \ell$. We have

$$W_j^\ell = \sum_{r=1}^{n_{\ell+1}} w(v_j^\ell, v_r^{\ell+1}) W_r^{\ell+1} = \sum_{r=1}^{n_{\ell+1}} [\mathbf{A}_\ell]_{jr} \sum_{h=1}^{n_{k+1}} [\mathbf{A}_{\ell+1} \cdots \mathbf{A}_k]_{rh}$$

$$= \sum_{h=1}^{n_{k+1}} \sum_{r=1}^{n_{\ell+1}} [\mathbf{A}_\ell]_{jr} [\mathbf{A}_{\ell+1} \cdots \mathbf{A}_k]_{rh} = \sum_{h=1}^{n_{k+1}} [\mathbf{A}_\ell \cdots \mathbf{A}_k]_{jh} \ .$$

∎

Proposition 2.3 *The probability that a random walk from node v_j^ℓ at layer ℓ ($1 \le \ell \le k$) terminates at the node v_i^{k+1} is $\frac{[\mathbf{A}_\ell \cdots \mathbf{A}_k]_{ji}}{\sum_{h=1}^{n_{k+1}} [\mathbf{A}_\ell \cdots \mathbf{A}_k]_{jh}}$.*

Proof: By downward induction on ℓ. Immediate for $\ell = k$: The probability assigned to an edge (v_j^k, v_i^{k+1}) equals $\frac{[\mathbf{A}_k]_{ji}}{\sum_{h=1}^{n_{k+1}} [\mathbf{A}_k]_{jh}}$. Assume the proposition holds for $\ell > r$. The probability that a random walk from v_j^r ends at v_i^{k+1} equals

$$\sum_{q=1}^{n_{r+1}} p_{jq}^r \frac{[\mathbf{A}_{r+1} \cdots \mathbf{A}_k]_{qi}}{\sum_h [\mathbf{A}_{r+1} \cdots \mathbf{A}_k]_{qh}} .$$

Proposition 2.2 asserts that

$$p_{jq}^r = [\mathbf{A}_r]_{jq} \frac{W_q^{r+1}}{W_j^r} = [\mathbf{A}_r]_{jq} \frac{\sum_{h=1}^{n_{k+1}} [\mathbf{A}_{r+1} \cdots \mathbf{A}_k]_{qh}}{\sum_{h=1}^{n_{k+1}} [\mathbf{A}_r \cdots \mathbf{A}_k]_{jh}} .$$

Hence, the probability that the walk terminates at v_i^{k+1} is

$$\frac{\sum_{q=1}^{n_{r+1}} [\mathbf{A}_r]_{jq} [\mathbf{A}_{r+1} \cdots \mathbf{A}_k]_{qi}}{\sum_{h=1}^{n_{k+1}} [\mathbf{A}_r \cdots \mathbf{A}_k]_{jh}}$$
$$= \frac{[A_r \cdots A_k]_{ji}}{\sum_{h=1}^{n_{k+1}} [\mathbf{A}_r \cdots \mathbf{A}_k]_{jh}} .$$

This conclude the proof. ∎

2.4. Accuracy and variance of scores

Consider some row $i \in \{1, \ldots, n_1\}$ of $\mathbf{P}$. Let

$$p_j = \frac{[\mathbf{P}]_{ij}}{\sum_{h=1}^{n_{k+1}} [\mathbf{P}]_{ih}} = \frac{[\mathbf{P}]_{ij}}{M_i}$$

denote the probability that a single sample is j. S_j has a binomial distribution with parameters S and p_j. The expected value of S_j is Sp_j. The probability that $S_j = t$ (j is sampled t times) is

$$\binom{S}{t} p_j^t (1 - p_j)^{S-t} .$$

(see, *e.g.* Feller[15] for background.) Using the normal approximation,

$$\begin{aligned} \mathrm{Prob}\{S_j \ge T\} &\simeq 1 - \Phi\left(\frac{T - 0.5 - Sp_j}{\sqrt{Sp_j(1-p_j)}}\right) \\ \mathrm{Prob}\{S_j < T\} &\simeq \Phi\left(\frac{T + 0.5 - Sp_j}{\sqrt{Sp_j(1-p_j)}}\right) \end{aligned} \tag{1}$$

The asymptotic tail behavior is given by additive and multiplicative forms of the Chernoff bounds [6]:

$$\begin{aligned} \mathrm{Prob}\left\{\frac{S_j}{S} > p_j + \epsilon\right\} &\le e^{-2S\epsilon^2} \\ \mathrm{Prob}\left\{\frac{S_j}{S} < p_j - \epsilon\right\} &\le e^{-2S\epsilon^2} \end{aligned} \tag{2}$$

$$\begin{aligned} \mathrm{Prob}\left\{\frac{S_j}{S} > p_j(1 + \epsilon)\right\} &\le e^{-Sp_j\epsilon^2/3} \\ \mathrm{Prob}\left\{\frac{S_j}{S} < p_j(1 - \epsilon)\right\} &\le e^{-Sp_j\epsilon^2/2} \end{aligned} \tag{3}$$

Bounds on additive error.

It follows from (2) that for each column j, the estimation error of the estimate $\frac{S_j}{S}M_i$ for $[\mathbf{P}]_{ij}$ is bounded by

$$\text{Prob}\left\{\left|\frac{S_j}{S}M_i - [\mathbf{P}]_{ij}\right| \geq \epsilon M_i\right\} \leq 2e^{-2S\epsilon^2} \ . \tag{4}$$

Hence, for the objective of estimating *all* entries in the ith row of $\mathbf{P}$ to accuracy $\pm\alpha$ with confidence at least $1 - 1/\operatorname{poly}(n_{k+1})$, we need

$$S = O\left(\left(\frac{M_i}{\alpha}\right)^2 \log n_{k+1}\right) \tag{5}$$

samples. Therefore, the total number of samples needed to estimate, with confidence $1-1/\operatorname{poly}(n_1, n_{k+1})$, all entries of the matrix $\mathbf{P}$ to within an additive error of $\pm\alpha$ is

$$O\left(\frac{\sum_{i=1}^{n_1}(M_i)^2}{\alpha^2}\log(n_1 + n_{k+1})\right) \ . \tag{6}$$

We obtain the following:

Proposition 2.4 *Let $\mathbf{A}_1, \ldots, \mathbf{A}_k$ be k nonnegative matrices. Let $\mathbf{A}^{n_1 \times n_2} = \mathbf{A}_1 \cdots \mathbf{A}_k$. Denote by N the number of nonzero entries in the matrices $\mathbf{A}_1, \ldots, \mathbf{A}_k$. For any $\alpha > 0$, in time*

$$O\left(N + k\left(\frac{\sum_{i=1}^{n_1}(M_i)^2}{\alpha^2}\log(n_1 + n_2)\right)\right) \ ,$$

with confidence $1 - 1/\operatorname{poly}(n_1, n_2)$, our sampling algorithm produces a matrix $\hat{\mathbf{A}}$ such that

$$\forall (i,j) \in \{1, \ldots, n_1\} \times \{1, \ldots, n_2\}, \ \left|[\mathbf{A}]_{ij} - [\hat{\mathbf{A}}]_{ij}\right| \leq \alpha \ .$$

Bounds on relative error.

From (3) we obtain:

$$\text{Prob}\left\{\left|\frac{S_j}{S}M_i - [\mathbf{P}]_{ij}\right| \geq \epsilon[\mathbf{P}]_{ij}\right\} \leq 2e^{-Sp_j\epsilon^2/3} \ . \tag{7}$$

Note that the relative error depends on the magnitude of the entry and decreases for larger entries. From Equation 7 we obtain:

Proposition 2.5 *For any $1 \geq p > 0$, $O(p^{-1}\epsilon^{-2}\log n_{k+1})$ samples suffice to guarantee that with confidence at least $(1-1/\operatorname{poly}(n_{k+1}))$, all columns j in the ith row are estimated within the following*

$$\forall j \begin{cases} \frac{|\hat{m}_j - m_j|}{m_j} & \leq & \epsilon\sqrt{\frac{p}{p_j}} & \textit{(relative error)} \\ |\hat{m}_j - m_j| & \leq & \epsilon M\sqrt{pp_j} & \textit{(additive error)} \end{cases}$$

The objective in many applications is to isolate and estimate all entries with value that exceeds a certain threshold. The following Corollary considers a threshold pM. All entries with value above pM are estimated with relative error at most ϵ. All entries smaller than pM by at least a fraction are estimated to be less than pM.

Corollary 2.6 *For any* $1 \geq p > 0$, $O(p^{-1}\epsilon^{-2}\log n_{k+1})$ *samples guarantee that with confidence at least* $(1 - 1/\operatorname{poly}(n_{k+1}))$, *all columns* j *in the* i*th row are estimated as follows:*

$$\forall j \textit{ such that } p_j \geq p, \ \left|\frac{S_j}{S}M_i - [\mathbf{P}]_{ij}\right| \leq \epsilon[\mathbf{P}]_{ij} \ .$$

Moreover,

$$\forall j, \ \ \frac{S_j}{S}M_i < p(1-\epsilon)^{1.5} \textit{ implies } p_j < p \ .$$

The largest entry in the ith row must have $p_j \geq 1/n_{k+1}$. It follows that

Corollary 2.7 $O(n_{k+1}\epsilon^{-2}\log n_{k+1})$ *samples guarantee that with confidence at least* $(1-1/\operatorname{poly}(n_{k+1}))$, *all entries larger than the mean (particularly, the largest entry) in the row are estimated with relative error at most* ϵ.

In the text retrieval datasets we experimented with, the largest entries in a row often had a much larger fraction of the total row weight ($p_j \gg 1/n$). Hence, even fewer samples suffice to guarantee ϵ relative error for the largest entries.

2.5. An alternative approach

We comment on a different and conceptually simpler approximate multiplication algorithm. The algorithm randomly samples $d' \ll d$ coordinates and performs the dot products computations on these coordinates only, yielding approximate results. This algorithm is faster than the exact computation by a factor of $\frac{d}{d'}$. The actual dot products can be approximated by multiplying the corresponding values of the abbreviated computation by $\frac{d}{d'}$. The expected value of these estimates equals the actual values, as in our method. However, the variance is typically much higher, in particular when most of the contribution to a high dot products is from a few coordinates. In this method, the variance of the ratio of the estimate and true value, depends on the weight distribution of the instance vectors and therefore, is unpredictable (dot products with the same exact value could have estimates with very different variance).

2.6. Dynamically changing database

So far, we viewed the matrices $\mathbf{A}_2, \ldots, \mathbf{A}_k$ as a static database, and allowed only the rows of $\mathbf{A}_1$ to be provided online. Here we consider the handling of a stream of queries, when rows, columns, or single entries, can be updated in the matrices $\mathbf{A}_2, \ldots, \mathbf{A}_k$.

The problem boils down to the cost of maintaining an updated version of the transition graph. Hence, a lower bound for the cost of updating the graph is linear time in the number of transition probabilities that need to be modified. This lower bound is always met, but the required number of modifications is sometimes much larger than the number of entries changed.

If an entry $[\mathbf{A}_i]_{hj}$ is updated, we need to update the transition probabilities on all outgoing edges, from every node in layers $1, \ldots, i$ that can reach v_h^i. The cost is linear in the number of outgoing edges incident to the reachability set of v_h^i in the reversed graph. Note that the cost of updating the whole row $[\mathbf{A}]_{h\bullet}$ is the same as updating a single entry in the row.

As is typical in dynamic algorithms, we can obtain a tradeoff between the cost of maintaining an updated transition graph and the query cost. That is, update cost can be decreased, by not performing the full explicit update, but then the query time increases.

3. Sampling for real-valued matrix products

We generalize the sampling algorithm of Section 2 to products of real-valued matrices.

3.1. Positive and negative contributions of matrix products

The product of real-valued matrices can be presented as the difference of two quantities, the *positive* and the *negative* contributions of the product. Each contribution can be estimated using a modification of our sampling algorithm.

Definition 3.1 Let $\mathbf{B}_1^{n_1\times n_2},\ldots,\mathbf{B}_i^{n_i\times n_{i+1}},\ldots,\mathbf{B}_k^{n_k\times n_{k+1}}$ be matrices with real entries. The *positive contribution* $\overbrace{\mathbf{B}_1\cdots\mathbf{B}_k}^{+}$ and the *negative contribution* $\overbrace{\mathbf{B}_1\cdots\mathbf{B}_k}^{-}$ of the product $\mathbf{B}_1\cdots\mathbf{B}_k$ are matrices of dimension $n_1\times n_{k+1}$ and are defined inductively as follows. If $k=1$ then

$$
\begin{aligned}
[\overbrace{\mathbf{B}_1}^{+}]_{ij} &= \begin{cases} [\mathbf{B}_1]_{ij} & \text{if } [\mathbf{B}_1]_{ij} > 0 \\ 0 & \text{otherwise} \end{cases} \\
[\overbrace{\mathbf{B}_1}^{-}]_{ij} &= \begin{cases} -[\mathbf{B}_1]_{ij} & \text{if } [\mathbf{B}_1]_{ij} < 0 \\ 0 & \text{otherwise} \end{cases}
\end{aligned}
$$

For $k>1$,

$$
\begin{aligned}
[\overbrace{\mathbf{B}_1\cdots\mathbf{B}_k}^{+}]_{ij} &= \sum_{h=1}^{n_2}[\overbrace{\mathbf{B}_1}^{+}]_{ih}[\overbrace{\mathbf{B}_2\cdots\mathbf{B}_k}^{+}]_{hj} + \sum_{h=1}^{n_2}[\overbrace{\mathbf{B}_1}^{-}]_{ih}[\overbrace{\mathbf{B}_2\cdots\mathbf{B}_k}^{-}]_{hj} \\
[\overbrace{\mathbf{B}_1\cdots\mathbf{B}_k}^{-}]_{ij} &= \sum_{h=1}^{n_2}[\overbrace{\mathbf{B}_1}^{+}]_{ih}[\overbrace{\mathbf{B}_2\cdots\mathbf{B}_k}^{-}]_{hj} + \sum_{h=1}^{n_2}[\overbrace{\mathbf{B}_1}^{-}]_{ih}[\overbrace{\mathbf{B}_2\cdots\mathbf{B}_k}^{+}]_{hj}\ .
\end{aligned}
$$

Each entry of a matrix product of k matrices can be viewed as a sum of products of k scalars. The positive contribution is the sum of all positive scalar products and the negative contribution is the (negated) sum of all negative scalar products.

Proposition 3.2 *For any k matrices $\mathbf{B}_1,\ldots,\mathbf{B}_k$ we have,*

$$
\mathbf{B}_1\cdots\mathbf{B}_k = \overbrace{\mathbf{B}_1\cdots\mathbf{B}_k}^{+} - \overbrace{\mathbf{B}_1\cdots\mathbf{B}_k}^{-} \tag{8}
$$

$$
\overbrace{\mathbf{B}_1\cdots\mathbf{B}_k}^{+} + \overbrace{\mathbf{B}_1\cdots\mathbf{B}_k}^{-} = \left(\overbrace{\mathbf{B}_1}^{+} + \overbrace{\mathbf{B}_1}^{-}\right)\cdots\left(\overbrace{\mathbf{B}_k}^{+} + \overbrace{\mathbf{B}_k}^{-}\right) \tag{9}
$$

Proof: By induction on k. ∎

3.2. The modified sampling algorithm

Consider a product

$$\mathbf{P}^{n_1 \times n_{k+1}} = \mathbf{A}_1^{n_1 \times n_2} \cdots \mathbf{A}_k^{n_k \times n_{k+1}}$$

of $k \geq 2$ real-valued matrices.

The sampling algorithm preprocesses these matrices in linear time in the number of nonzero entries. It generates a $k+1$ layer directed graph whose edges are labeled with transition probabilities. The first layer has $2n_1$ nodes and the $(k+1)$st layer has n_{k+1} nodes. Random walks from the first layer to the $(k+1)$st layer corresponds to sampling in the positive or negative contributions of the product. The layered graph is constructed as follows.

- **Nodes:** For $1 \leq i \leq k$, the ith layer has $2n_i$ nodes. Let

$$\begin{aligned} V_i^+ &= \{v_1^{i+}, v_2^{i+}, \ldots, v_{n_i}^{i+}\} \\ V_i^- &= \{v_1^{i-}, v_2^{i-}, \ldots, v_{n_i}^{i-}\} \end{aligned}$$

  Let $V_i = V_i^+ \cup V_i^-$ be the set of nodes at layer i. The $k+1$st layer consists of n_{k+1} nodes $V_{k+1} = \{v_1^{k+1}, \ldots, v_{n_{k+1}}^{k+1}\}$.

- **Edges:** For $i < k$, the edges between the ith and the $(i+1)$st layers are as follows:

  – Edges in $V_i^+ \times V_{i+1}^+$ and edges in $V_i^- \times V_{i+1}^-$ are as in the bipartite graph $G(\overbrace{\mathbf{A}_i}^{+})$.

  – Edges in $V_i^+ \times V_{i+1}^-$ and edges in $V_i^- \times V_{i+1}^+$ are as in the bipartite graph $G(\overbrace{\mathbf{A}_i}^{-})$.

  For $i = k$,

  – edges in $V_k^+ \times V_{k+1}$ are as in $G(\overbrace{\mathbf{A}_k}^{+})$ and

  – edges in $V_k^- \times V_{k+1}$ are as in $G(\overbrace{\mathbf{A}_k}^{-})$.

The preprocessing step assigns transition probabilities $p : E$ to the edges of the graph such that a random walk from v_i^{1+} to the $(k+1)$st layer corresponds to a sample from row i of the positive contribution of the product. The index of the last node reached is j with probability

$$\frac{[\overbrace{\mathbf{A}_1 \cdots \mathbf{A}_k}^{+}]_{ij}}{\sum_{h=1}^{n_{k+1}} [\overbrace{\mathbf{A}_1 \cdots \mathbf{A}_k}^{+}]_{ih}}.$$

Similarly, a random walk from v_i^{1-} corresponds to a sample from row i of the negative contribution of the product. The index of the last node reached is j with probability

$$\frac{[\overbrace{\mathbf{A}_1 \cdots \mathbf{A}_k}^{-}]_{ij}}{\sum_{h=1}^{n_{k+1}} [\overbrace{\mathbf{A}_1 \cdots \mathbf{A}_k}^{-}]_{ih}}.$$

The motivation for doubling of the nodes sets (use of V_i^- and V_i^+) in layers $i \in \{1,\ldots,k\}$ is to keep track of the sign of the scalar product corresponding to the current random walk.

The transition probabilities are computed by first computing, for every $1 \le i \le k$ and $1 \le j \le n_i$, the sums

$$\begin{aligned} W_j^{i+} &= \sum_{h=1}^{n_{k+1}} [\overbrace{\mathbf{A}_i \cdots \mathbf{A}_k}^{+}]_{jh} \\ W_j^{i-} &= \sum_{h=1}^{n_{k+1}} [\overbrace{\mathbf{A}_i \cdots \mathbf{A}_k}^{-}]_{jh} \end{aligned}$$

(the weight of the ith rows of $\overbrace{\mathbf{A}_j \cdots \mathbf{A}_k}^{+}$ and $\overbrace{\mathbf{A}_j \cdots \mathbf{A}_k}^{-}$, respectively). We associate these sums with the corresponding nodes and denote $W_j^{i+} \equiv W(v_j^{i+})$ and $W_j^{i-} \equiv W(v_j^{i-})$.

The following is a modification of Algorithm 2.1.

Algorithm 3.3 [Compute transition probabilities]

- Compute W_j^{i+}, W_j^{i-} :
  1. For all $v \in V_{k+1}$, set $W(v) \leftarrow 1$ (layer $k+1$ nodes).
  2. Compute $W : V_k$ (layer k nodes): For $1 \le j \le n_k$, set

     $$\begin{aligned} W_j^{k+} &\leftarrow \sum_{h=1}^{n_{k+1}} [\overbrace{\mathbf{A}_k}^{+}]_{jh} \\ W_j^{k-} &\leftarrow \sum_{h=1}^{n_{k+1}} [\overbrace{\mathbf{A}_k}^{-}]_{jh} \end{aligned}$$

     (the sum of the weights of outgoing edges from v_j^{k+} or v_j^{k-} to V^{k+1}).
  3. For $i = k-1$ down to 1: (compute $W : V_i$)
     For every node $v \in V_i$, set $W(v)$ to the sum of the products $w(v,u)W(u)$, for all $u \in V^{i+1}$ such that (v,u) is an edge.

- Assign transition probabilities:
  For every edge $e = (v,u)$, set

  $$p(e) = \frac{w(e)W(u)}{W(v)} \, .$$

Interestingly, the estimation of the positive and negative contributions is intertwined. The graph can not be split to two separate graphs, each with n_i nodes at layer i $(i = 1,\ldots,k+1)$ for estimating the positive and negative contributions.

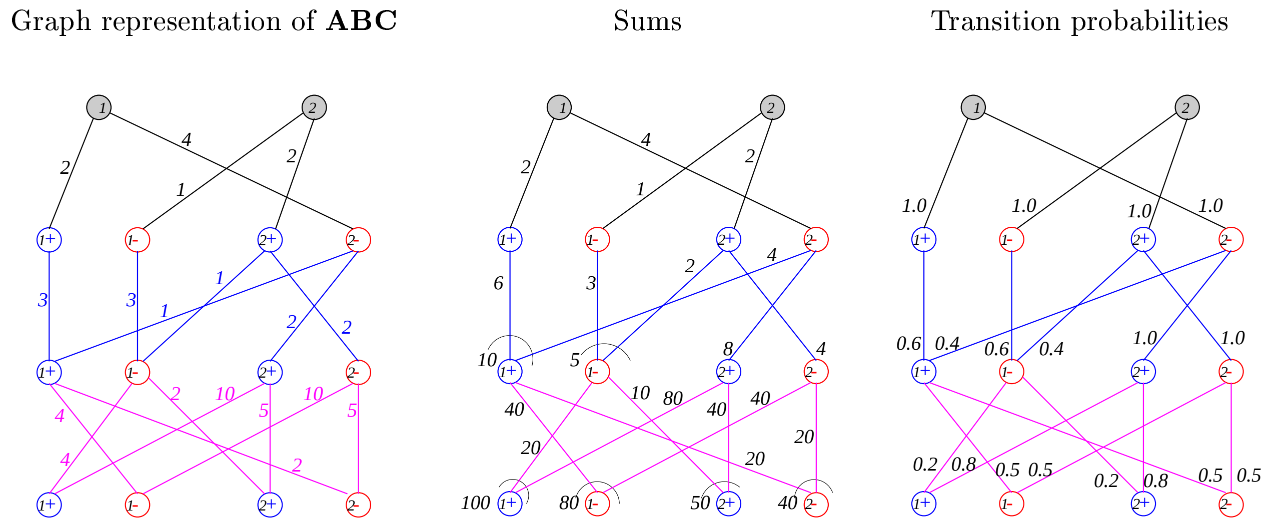

Figure 2: The product graph and probability assignment

3.3. Example

As an example, consider the 2×2 matrices $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$:

$$\mathbf{A} = \begin{array}{|c|c|}\hline -4 & +10 \\ \hline -2 & +5 \\ \hline\end{array} \quad \mathbf{B} = \begin{array}{|c|c|}\hline +3 & -1 \\ \hline 0 & -2 \\ \hline\end{array} \quad \mathbf{C} = \begin{array}{|c|c|}\hline +2 & -1 \\ \hline -4 & +2 \\ \hline\end{array}$$

The value, positive, and negative contribution of the subproducts $\mathbf{AB}$, $\mathbf{BC}$ and $\mathbf{ABC}$ are as follows:

$$\begin{array}{lclclcl}
\mathbf{BC} & = & \begin{array}{|c|c|}\hline 10 & -5 \\ \hline 8 & -4 \\ \hline\end{array} & \overbrace{\mathbf{BC}}^{+} & = \begin{array}{|c|c|}\hline 10 & 0 \\ \hline 8 & 0 \\ \hline\end{array} & \overbrace{\mathbf{BC}}^{-} & = \begin{array}{|c|c|}\hline 0 & 5 \\ \hline 0 & 4 \\ \hline\end{array} \\
\mathbf{AB} & = & \begin{array}{|c|c|}\hline -12 & -16 \\ \hline -6 & -8 \\ \hline\end{array} & \overbrace{\mathbf{AB}}^{+} & = \begin{array}{|c|c|}\hline 0 & 4 \\ \hline 0 & 2 \\ \hline\end{array} & \overbrace{\mathbf{AB}}^{-} & = \begin{array}{|c|c|}\hline 12 & 20 \\ \hline 6 & 10 \\ \hline\end{array} \\
\mathbf{ABC} & = & \begin{array}{|c|c|}\hline +40 & -20 \\ \hline +20 & -10 \\ \hline\end{array} & \overbrace{\mathbf{ABC}}^{+} & = \begin{array}{|c|c|}\hline 80 & 20 \\ \hline 40 & 10 \\ \hline\end{array} & \overbrace{\mathbf{ABC}}^{-} & = \begin{array}{|c|c|}\hline 40 & 40 \\ \hline 20 & 20 \\ \hline\end{array}
\end{array}$$

Figure 2 illustrates, from left to right, the product graph with edge weights, the graph with the values of W_j^{i+} and W_j^{i-} on the nodes and $w(v,u)W(u)$ on each edge (u,v), and the graph with transition probabilities assigned to edges.

3.4. Correctness

The following propositions assert correctness. Their proofs are similar, but notationally more involved, than the proofs of Propositions 2.2 and 2.3. They are left as an exercise for the reader.

Proposition 3.4 *For $i = 1, \ldots, k$ and $1 \le j \le n_i$,*

$$W_j^{i+} \quad = \quad \sum_{h=1}^{n_{k+1}} [\overbrace{\mathbf{A}_i \cdots \mathbf{A}_k}^{+}]_{jh}$$

$$W_j^{i-} = \sum_{h=1}^{n_{k+1}} [\overbrace{\mathbf{A}_i \cdots \mathbf{A}_k}^{-}]_{jh}$$

Proposition 3.5 *The probability that a random walk from node v_j^{1+} (resp., v_j^{1-}) terminates at the node v_i^{k+1} is*

$$\frac{[\overbrace{\mathbf{A}_1 \cdots \mathbf{A}_k}^{+}]_{ji}}{\sum_{h=1}^{n_{k+1}} [\overbrace{\mathbf{A}_1 \cdots \mathbf{A}_k}^{+}]_{jh}} .$$

(respectively,

$$\frac{[\overbrace{\mathbf{A}_1 \cdots \mathbf{A}_k}^{-}]_{ji}}{\sum_{h=1}^{n_{k+1}} [\overbrace{\mathbf{A}_1 \cdots \mathbf{A}_k}^{-}]_{jh}} .)$$

3.5. Estimation accuracy

The analysis in Subsection 2.4 applies for estimates of either the positive or the negative contributions of the product. Our relative error bounds diverge when the two quantities are close in value, since the actual dot product values are the difference of these quantities.

Denote

$$M'^{+}_i = \sum_{h=1}^{n_2} [\overbrace{\mathbf{A}_1 \cdots \mathbf{A}_k}^{+}]_{ih}$$

$$M'^{-}_i = \sum_{h=1}^{n_2} [\overbrace{\mathbf{A}_1 \cdots \mathbf{A}_k}^{-}]_{ih}$$

$$p_j^{+} = \frac{[\overbrace{\mathbf{A}_1 \cdots \mathbf{A}_k}^{+}]_{ij}}{M'^{+}_i}$$

$$p_j^{-} = \frac{[\overbrace{\mathbf{A}_1 \cdots \mathbf{A}_k}^{-}]_{ij}}{M'^{-}_i}$$

The following additive bound is a corollary of Proposition 2.4.

Proposition 3.6 *Let $\mathbf{A}_1, \ldots, \mathbf{A}_k$ be k real-valued matrices. Let $\mathbf{A}^{n_1 \times n_2} = \mathbf{A}_1 \cdots \mathbf{A}_k$. For $i = 1, \ldots, n_1$, let $M'_i = M'^{+}_i + M'^{-}_i$. Denote by N the number of nonzero entries in the matrices $\mathbf{A}_1, \ldots, \mathbf{A}_k$. For any $\alpha > 0$, in time*

$$O\left(N + k\left(\frac{\sum_{i=1}^{n_1}(M'_i)^2}{\alpha^2}\log(n_1+n_2)\right)\right) ,$$

the modified sampling algorithm produces a matrix $\hat{\mathbf{A}}$ such that

$$\forall (i,j) \in \{1,\ldots,n_1\} \times \{1,\ldots,n_2\}, \ \left|[\mathbf{A}]_{ij} - [\hat{\mathbf{A}}]_{ij}\right| \le \alpha .$$

Proof: It is sufficient to obtain additive error $\alpha/2$ on both the positive and negative contributions of the product. It is easy to see that $O(\left(\frac{\sum_{i=1}^{n_1}(M'^{+}_i)^2}{\alpha^2}\log(n_1+n_2)\right))$ samples suffice for the positive contribution and $O(\left(\frac{\sum_{i=1}^{n_1}(M'^{-}_i)^2}{\alpha^2}\log(n_1+n_2)\right))$ samples suffice for the negative contribution. ∎

The following is a corollary of Proposition 2.5.

Proposition 3.7 *For any* $1 \geq p > 0$, *using* $O(p^{-1}\epsilon^{-2}\log n_{k+1})$ *samples for estimating the negative and positive contributions yields confidence of at least* $(1 - 1/\operatorname{poly}(n_{k+1}))$ *for*

$$\forall j \begin{cases} \frac{|\hat{m}_j - m_j|}{m_j} & \leq & \epsilon\sqrt{p}\left(\frac{M'^{+}\sqrt{p_j^{+}}+M'^{-}\sqrt{p_j^{-}}}{M'^{+}p_j^{+}-M'^{-}p_j^{-}}\right) & \textit{(relative error)} \\ |\hat{m}_j - m_j| & \leq & \epsilon\sqrt{p}\left(M'^{+}\sqrt{p_j^{+}} + M'^{-}\sqrt{p_j^{-}}\right) & \textit{(additive error)} \end{cases}$$

4. Implementation issues

We implemented the sampling algorithm for products of $k = 2$ nonnegative matrices. The input consists of a "database" matrix $\mathbf{A}^{d\times n}$, where n is the number of instances and d is the number of features. Query vectors of dimension d are either provided beforehand or presented online. For each input query $\mathbf{q} \in \mathcal{R}^d$ and a parameter S (number of samples), the algorithm computes $M = \sum_{i=1}^{n}[\mathbf{q}^T\mathbf{A}]_i$ and a (possibly sparse) integral *score vector* $(S_1, \ldots, S_n)$, where S_i is the number of times i was sampled.

4.1. Implementing the sampling algorithm

We provided efficient implementations for both dense and sparse matrices. Sparsity is fully exploited in the sense that we exhibit considerable speedups over sparse matrix multiplication algorithms.

Presampling.

A useful implementation strategy is to precompute and store samples based on the matrix product graph, rather than storing the graph itself. The samples are generated in the preprocessing of the database matrix $\mathbf{A}$. We generated lists $L_1, \ldots, L_d$ of indices $\{1, \ldots, n\}$ of sampled instances. For each input query $\mathbf{q}$, the algorithm processes $\mathbf{q}$ and determines how many samples are needed from each of the d lists $L_1, \ldots, L_d$. It then retrieves the appropriate number of samples from each stored list and counts how many times each of the indices $\{1, \ldots, n\}$ appears. *Presampling* improves performance because the same samples are used for different queries. Hence, while evaluating a query, each sampled instance requires only a memory lookup instead of the generation of a random number. Furthermore, presampling is effective from a memory management standpoint, since instances in the sample are grouped together on disk or in memory (rather than dispersed throughout a large database).

In the preprocessing stage we select the list size $\ell_i = |L_i|$ $(i = 1, \ldots n)$ to be the maximum number of samples from L_i we expect to be required by any query. In our experiments we selected the ℓ_i's to be the maximum required for the desired accuracy when the database instances themselves are used as queries. For our data, the sample-lists required considerably less storage than the original database matrix (or its graph representation). We should note that presampling introduces correlations between results of independent queries (i.e. certain instances will systematically have higher or lower scores for similar queries). In particular, the same instance will always have the same score for the same query, which is in fact desirable in some applications. However, if desired, the correlation of scores across queries can be reduced by using larger presamples and by using randomization in the selection within each list.

The use of presampling calls for a different implementation strategy than the straightforward implementation of the algorithm of Section 2. We sketch below the modified preprocessing and query processing stages. The particulars of the implementation vary for handling sparse or dense data efficiently, and should be obvious to the reader from the description below.

Preprocessing: Generating the sample lists.

- For each row $1 \le i \le d$ of $\mathbf{A}$:
  Compute the sum $a_i \leftarrow \sum_h [\mathbf{A}]_{ih}$.
  Initialize L_i to the empty list.
- Repeat for ℓ_i times (desired number of samples):
  Draw an instance $j \in \{1, \ldots, n\}$ at random, where j has probability $\frac{[\mathbf{A}]_{ij}}{a_i}$, and append it to the list L_i.

Process a query vector q (with S samples).

1. For every $1 \le i \le d$ such that $q_i \neq 0$: let $q'_i \leftarrow q_i a_i$; let $M \leftarrow \sum_{i=1}^{d} q'_i$ $(= \sum_{h=1}^{n} [\mathbf{q}^T \mathbf{A}]_h)$.
2. For each $1 \le i \le d$, determine the number of samples c_i needed from the list L_i. $(S = \sum_i c_i.)$
   We choose c_i such that it has expected value $\frac{S q'_i}{\sum_{j=1}^{d} q'_j}$:
$$c_i = \begin{cases} \left\lfloor \frac{S q'_i}{\sum_{j=1}^{d} q'_j} \right\rfloor & \text{with probability } \left\lceil \frac{S q'_i}{\sum_{j=1}^{d} q'_j} \right\rceil - \frac{S q'_i}{\sum_j q'_j} \\ \left\lceil \frac{S q'_i}{\sum_{j=1}^{d} q'_j} \right\rceil & \text{with probability } \frac{S q'_i}{\sum_{j=1}^{d} q'_j} - \left\lfloor \frac{S q'_i}{\sum_{j=1}^{d} q'_j} \right\rfloor \end{cases}$$
3. **Scoring:** For $i = 1, \ldots, d$:
   Repeat c_i times:
   - Let $s \in \{1, \ldots, n\}$ be the next sample in L_i.
   - If s is encountered for the first time: create a counter for s with value 1.
     Otherwise, increment the counter of s.
4. For $j \in \{1, \ldots, n\}$, set S_j (the score of j) to the value of the counter (if initialized), and 0 otherwise.

4.2. Chain products and real-valued data

Our implementation ideas extend to $k > 2$ matrices and real-valued data. For real-valued data (see Section 3), we generate two sets of sample lists, one for the positive contribution and one for the negative contribution. In each query evaluation, we need to decide how many samples are taken for estimating each contribution. Issues in selecting the sample size are discussed later in this section for nonnegative data, and are similar for real-valued data.

When our "database matrix" $\mathbf{A}$ is represented as a product $\mathbf{A} = \mathbf{A}_2 \cdots \mathbf{A}_k$ ($k > 2$), the only implementation difference is in the presampling stage. The sample lists are constructed with respect to the product $\mathbf{A}$ (using the method outlined in Section 2). The implementation of query processing and scoring is the same as the two-matrix case.

4.3. Using only scores to identify high matches

Consider a query $\mathbf{q}$ and the corresponding match vector $\mathbf{m} = \mathbf{q}^T\mathbf{A} = (m_1, \ldots, m_n)$. The expected value of S_i equals Sm_i/M (see Section 2). $M = \sum_{j=1}^{n} m_j$ is computed by the scoring algorithm. The expected value of S_iM/S equals m_i. Hence, high scores can be used to identify high matches. The variance of $\frac{\hat{m}_i}{m_i}$ decreases with the number of samples S and with the match value m_i (see Equation 1). We typically choose S according to tradeoffs between time and accuracy.

When query vectors and columns of $\mathbf{A}$ are nonnegative and normalized to have a Euclidean norm of 1.0, the m_i's are cosines in the range $[0, 1]$. To identify instances where $m_i \geq \alpha$ we use a threshold $t = \alpha(1 - \epsilon)$ and consider instances where $S_iM/S > t$. To ensure that the probability of being over or below the threshold depends only on the value of the entry and not on the particular row, we use a "global" parameter H (usually in the range $\{20, \ldots, 100\}$). For each query we use $S = HM$ samples. Choosing the threshold $T = tH$ implies $T > S_i$ if and only if $S_iM/S > t$. The values of H and the threshold T determine the probability that an entry with a certain value m_i scores above or below T. By Equation 1,

$$\begin{aligned} \text{Prob}\{S_i \geq T\} &\simeq 1 - \Phi\left(\frac{T - 0.5 - Hm_i}{\sqrt{Hm_i(1-m_i)}}\right) \\ \text{Prob}\{S_i < T\} &\simeq \Phi\left(\frac{T + 0.5 - Hm_i}{\sqrt{Hm_i(1-m_i)}}\right) \end{aligned} \tag{10}$$

Hence, appropriate values can be predetermined for objectives such as "Find a subset of $\{1, \ldots, n\}$ that (with high probability) contains at least 90% of all matches of value α and above and at most 5% of all matches of value $\alpha - 0.1$ and below". Note that predetermining the values of T and H assures that the desired property holds with respect to the "worst case" distribution of the unknown m_i's, where all the m_i equal either α or $\alpha - 0.1$.

4.4. Determining S online

For general distributions of the m_i's, some knowledge of the distribution allows us to make a better choice of H and T. Such knowledge can be obtained by increasing S online and looking at the intermediate distributions of scores to determine properties of the actual distribution. For example, if most values are much smaller or much larger than α, a smaller number of samples suffice.

The online approach is also preferable for identifying the k nearest neighbors of an instance. The distribution of m_i's as a whole, and in particular the distribution of the largest few m_i's (those for the nearest neighbors), will vary considerably from query instance to query instance.

Another example where it is better to determine S online is in identifying a set of instances I of a certain size L such that a *high fraction* of the instances in I have matches over some value α. An online algorithm that identifies such a set increases S (continues sampling), while maintaining the set I of L highest scoring items. The sampling stops when a test shows that a satisfactory fraction of the instances in I are high matches. The test can be based on the distribution of scores. Alternatively, the estimate of the fraction of high matches in I can be obtained by taking several uniform random samples from I and computing exact dot product values for the selected instances, while counting how many of these instances are high matches. Such a test can be performed each time the number of samples is increased by some factor γ (for example, $\gamma = 2$, each time S doubles in value). Hence, the test is performed $O(\log_\gamma S)$ times. The number of random samples from I needed for each test can be computed as a function of the desired fraction of high matches in I. This algorithm assures that the final number of samples S is at most γ times the "necessary" number. Such a guarantee can not be provided by an algorithm that predetermines S. Typically, there are many instances with low match values and a small fraction of them, which could still be a significant number, have high scores. This number decreases with S, but also depends on the distribution of the m_i's. Hence, predetermining S necessitates making worst case assumptions on the distribution which would often result in using many more samples than the online algorithm.

4.5. Using a combined algorithm

In many applications it is desirable to first compute (implicitly or explicitly) an approximate dot product value for all instances, followed by exactly computing dot products for a selected subset of the instances. One such scenario was given above, where exact computations were used to test the quality of our intermediary result, and decide whether more sampling is needed.

More generally, it is often desirable to efficiently identify the set of instances with high dot products, and then compute exact dot products only for those instances. To accomplish this, we first use sampling to filter out instances that are extremely unlikely to be high matches. Then we compute the dot products of $\mathbf{q}$ with the remaining instances, which typically constitute a much smaller set. This approach for obtaining exact high matches yields considerable speedups, particularly for dense data, over fully computing the product $\mathbf{q}^T\mathbf{A}$ (see Section 5).

Combined algorithm:.

- **Input:** A query $\mathbf{q}$ and parameters T (*threshold*) and H (*hit factor*).
- Apply the sampling algorithm with $S = HM$ samples to obtain scores $(S_1, \ldots, S_n)$.
- Let $I \subset \{1, \ldots, n\}$ be the subset of instances with scores above threshold $(S_i \geq T)$.
- Compute the dot product $\mathbf{q}^T\mathbf{x}_i$ for all $i \in I$ and identify the highest matches.

The choice of the parameters H and T determines a tradeoff between the size of the set I (the smaller the better), the number of high matches that are filtered out, and the cost of sampling (see Equation 10.) To obtain a small size set I, the threshold T should be such that T/H is larger than most values of $\{m_1, \ldots, m_n\}$. On the other hand, T and H should be such that T/H is below the magnitude of the lowest match which is of interest. The likelihood that matches above $T/H + \epsilon$ are filtered out, or values below $T/H - \epsilon$ are in the set I, decreases with S (see analysis in Section 2)

4.6. A heuristic for real-valued data

In Section 3 we proposed an algorithm to estimate real-valued matrix product by estimating the positive and negative contributions of the product. The actual dot-product value equals the value of the positive contribution minus the negative contribution. Here we propose a heuristic to identify large dot product values for real-valued matrices by estimating *only positive contributions.*

This heuristic was motivated by our observation that on the datasets we examined, high matches typically had high positive contribution and low negative contribution, and vice versa, high positive contribution implied high dot products. This relation holds when features are not independent, that is, similarity of some features of two instances, makes them more likely to be similar on other features. An intuitive explanation is that when two vectors are correlated, larger entries in corresponding columns are likely to have the same sign. The likelihood drops towards 50% when the two vectors are independent. We conjecture that this dependence is common in many other pattern recognition tasks. Our experiments (Section 5.1) demonstrate considerable speedups with negligible loss of accuracy for this heuristic.

Our heuristic is a "combined" algorithm (in the sense of Subsection 4.5). It first transforms vectors to a nonnegative form such that the dot product value for a pair of transformed vectors equals the positive contributions of the original dot-product. Following that, the sampling algorithm identifies vectors with high "transformed" dot products, and computes exact *original* dot products only for (the small set of) vectors with high "transformed" scores.

The following transformation is applied to the document and query vectors. For every such vector $\mathbf{q}$ we define $\tilde{\mathbf{q}}$ such that

$$\begin{aligned} \tilde{q}_{2i} &= \max\{0, q_i\} \\ \tilde{q}_{2i+1} &= -\min\{0, q_i\} \end{aligned} \tag{11}$$

Note that the vectors $\tilde{\mathbf{q}}$ are nonnegative. It is easy to see that the product $\tilde{\mathbf{q}} \cdot \tilde{\mathbf{x}}_i$ is the *positive contribution* of the product $\mathbf{q} \cdot \mathbf{x}_i$ (see Section 3). The sampling algorithm is applied to the nonnegative database matrix $\tilde{\mathbf{A}}^{n \times 2d}$ and nonnegative query vectors $\tilde{\mathbf{q}}$. It identifies (and scores) the objects with highest positive contributions. The exact (original) dot products are then explicitly computed for documents with high scores.

5. Experiments

As a test of our sampling method, we chose the task of estimating the largest entries in each row of $\mathbf{A}\mathbf{A}^T$, where each of the rows $\{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$ of $\mathbf{A}$ is the representation of a textual document as a set of numeric feature values. This finds, for each document, its set of nearest neighbors, under a dot product based proximity function. In information retrieval, such sets of nearest neighbors can be used to support hypertext browsing [24] or document clustering [26]. Finding nearest neighbors has similar computational characteristics to running large numbers of queries or other linear classifiers against a document database, and so is a good model task.

Different methods of representing texts will give document matrices with very different properties. We first describe the challenging case, where the representation yields a dense matrix of real-valued document vectors. Following that, we describe results for a sparse nonnegative representation. All experiments were run under similar load conditions using one processor of a SGI Challenge with 1 GB RAM.

5.1. Dense real-valued data

Several information retrieval approaches have been proposed which model the observed document-term matrix as having a simpler underlying structure based on document clusters, word clusters, or latent factors. The most popular such approach is Latent Semantic Indexing (LSI) [12]. LSI is based on Singular Value Decomposition (see [2] for background). The sparse document vectors $\mathbf{x}_i \in \mathcal{R}^t_+$ are replaced by dense d-dimensional vectors $\hat{\mathbf{x}}_i \in \mathcal{R}^d$ where $d \ll t$ (typically, d is of order 10^2). The coordinates of $\hat{\mathbf{x}}_i$ are linear combinations of the coordinates of $\mathbf{x}_i$, that is, the decomposition provides a transformation matrix $\mathbf{T} \in \mathcal{R}^{t \times d}$ such that $\hat{\mathbf{x}}_i = \mathbf{x}_i^T \mathbf{T}$ and hence, the new database matrix is $\hat{\mathbf{A}} = \mathbf{AT}$. The SVD decomposition guarantees that $\hat{\mathbf{A}}\hat{\mathbf{A}}^T$ is the best least-square approximation to $\mathbf{AA}^T$ that can be provided by a rank d matrix $\hat{\mathbf{A}} \in \mathcal{R}^{n \times d}$. Values of d of a few hundred typically give the best retrieval effectiveness, indicating that a limited rank approximation can be better content representation than the original matrix.

When using LSI for text retrieval, a user query is treated as a vector $\mathbf{q}$ of length t, and then transformed to a length d latent vector $\hat{\mathbf{q}}$ $(= \mathbf{q}^T\mathbf{T})$. The goal then is to find the set of objects $I \subset \{1, \ldots, n\}$ such that $\hat{\mathbf{q}} \cdot \hat{\mathbf{x}}$ is large. This again amounts to identifying the largest entries in the product $\hat{\mathbf{q}}^T \hat{\mathbf{A}}^T$.

For our experiments we used a collection of 5,000 encyclopedia articles, originally represented as a 5,000 by 75,714 sparse nonnegative matrix of term weights similar to those used in Section 5.2. The LSI procedure was applied to this matrix, producing a dense 5,000 document by 320 factor matrix $\hat{\mathbf{A}}$, which was then normalized so that the Euclidean norm of each document vector was 1.[2] We tested our sampling procedure by attempting to find all large document-document matches, in this case all large values in $\hat{\mathbf{A}}\hat{\mathbf{A}}^T$.

5.1.1. Dealing with negative entries

Even when (as is usual) the raw document-term matrix $\mathbf{A}$ is nonnegative, it is possible that $\mathbf{T}$ and thus $\hat{\mathbf{A}}$ contain negative entries. We applied the heuristic proposed in Subsection 4.6 to efficiently identify high dot products.

The underlying assumption for this heuristic is that there is a significant correlation between high dot product values and high positive contributions. We can show this correlation graphically. The relation between dot-product values and corresponding positive contributions for the *ency320.5000* dataset is plotted in Figures 3 and 4. Figure 3 plots the positive contribution vs. dot product value for the roughly 106,000 document pairs with dot product above 0.55 in the dataset *ency320.5000*. Figure 4 plots positive contribution vs. dot product for all pairs on a subset of 500 of the documents. In Figure 5 we plot this dependence for all pairs of 156 vectors from the *Isolet* data set (Isolated Letter Speech Recognition). We used normalized vectors with 617 attributes, where each vector corresponds to a voice sample of a spoken letter.[3]

[2]The LSI data was provided by Susan Dumais of Bellcore.

[3]This dataset is available from directory */pub/machine-learning-databases/isolet* on *ftp.ics.uci.edu*.

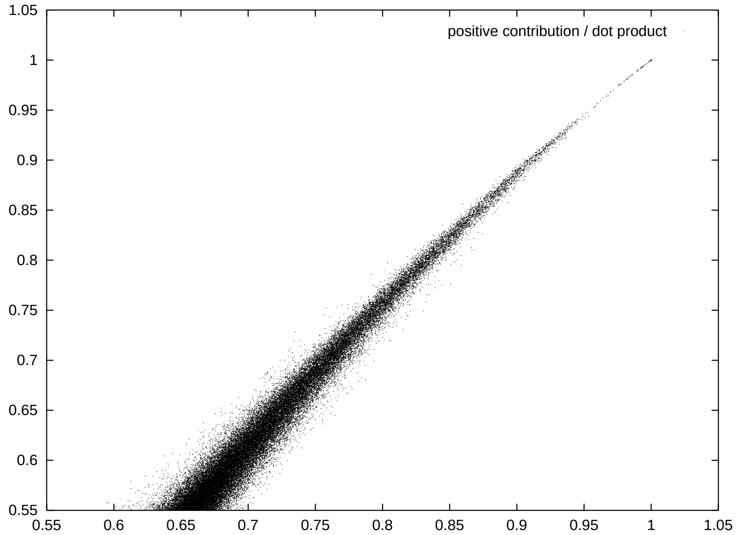

Figure 3: Plots of $(\hat{\mathbf{x}}_i \cdot \hat{\mathbf{x}}_j, \tilde{\mathbf{x}_i} \cdot \tilde{\mathbf{x}_j})$ for all document pairs such that $\hat{\mathbf{x}}_i \cdot \hat{\mathbf{x}}_j > 0.55$ (106K pairs).

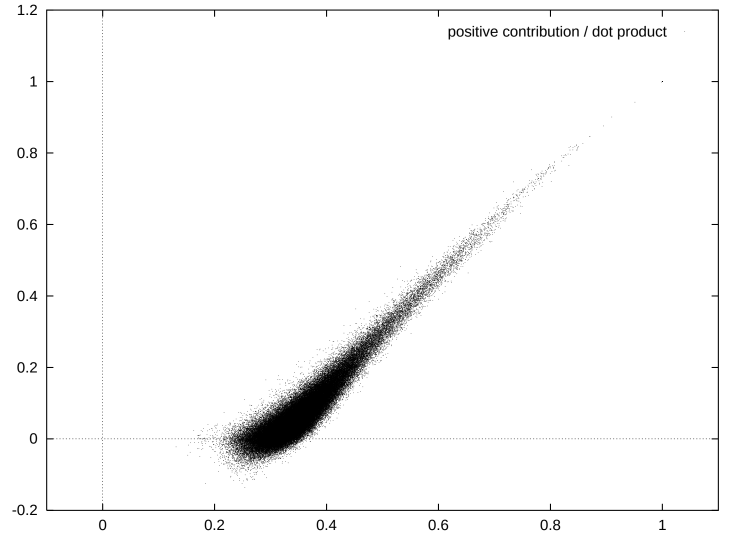

Figure 4: Plots of $(\hat{\mathbf{x}}_i \cdot \hat{\mathbf{x}}_j, \tilde{\mathbf{x}_i} \cdot \tilde{\mathbf{x}_j})$ for all pairs of 500 documents.

5.1.2. Results

We applied a combined sampling and partial dot products computation (as described in Section 4.6) to the dataset *ency320.5000*. We first applied a sampling algorithm with parameters H and T (hit-factor and threshold, as in Sections 4.3 and 4.5) on the positive matrix $\tilde{\mathbf{A}}$. That is, for a "query" $\tilde{\mathbf{q}}$ (a row of $\tilde{\mathbf{A}}^T$), we used $S = HM$ samples, where $M = \sum_{i=1}^{n}[\tilde{\mathbf{q}}^T\tilde{\mathbf{A}}^T]_i$. For all instances with score above T, we computed the exact dot product value $(\mathbf{q} \cdot \mathbf{x}_i)$.

Table 1 shows the results of this test. The first row in each box (algorithm *Full*) shows results for a naive algorithm that computes all dot products. We compare this to our combined sampling algorithm. The sampling algorithm has a one time cost to prepare the sampled representation ("Index" time), and the size of this representation ("Presamples Computed") is controlled by parameter H. The sampled representation is used to score all instances for each query ("Samples

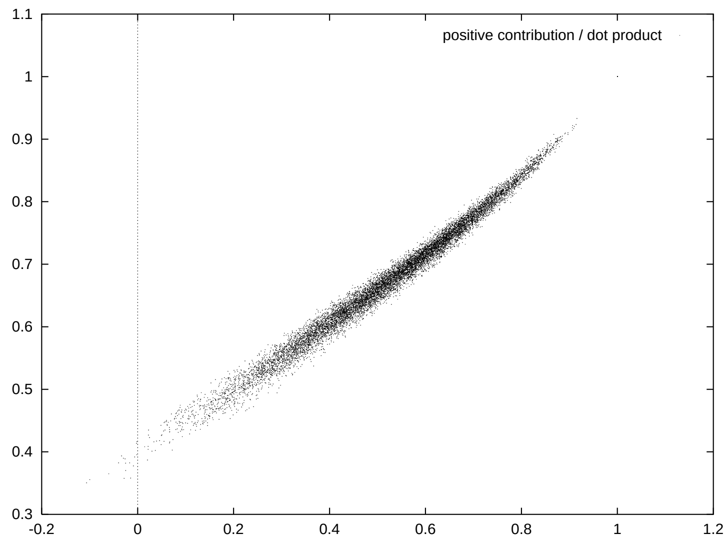

Figure 5: Plot for all positive/original dot products of 156 normalized vectors from *Isolet* data sets.

Drawn" is the total number of samples used at query time). Dot products are computed explicitly for instances with scores of at least T, so smaller values of T (for constant H) lead to more dot products being computed explicitly ("Dot Products Computed"). Each dot product computation amounts to 320 floating point multiply-adds, whereas each sampling step amounts to a memory lookup and a counter increment. The result is that the sampling algorithm can use much less compute time than *Full* while still identifying almost all high dot products. For example, Algorithm *R1* utilizes only 11% of the CPU time (9% if disregarding preprocessing) of *Full* and identifies 99.8%, 99.3%, 98.4%, 96.7%, 94.2% of the matches above 0.85, 0.80, 0.75, 0.70, and 0.65, respectively. Algorithm *R5* utilizes 20% of the CPU time of *Full* (18% if disregarding preprocessing) and identifies 99% of matches over 0.55.

We observed that most matches were in $[0, 1]$ and that the very few negative matches had small magnitudes. This is not surprising, since dot products of the LSI vectors should approximate the dot products of the (nonnegative) original vectors. Note that even though the scoring is with respect to the positive contributions of the dot products, the final dot products and the results in Table 1 are with respect to the actual values.

The number of samples in the sample matrices (computed in the preprocessing) are $738,801$ for *R1* and *R2*, $886,563$ for *R3* and *R4* and $1,182,037$ for *R1*. Recall that each sample is an identifier of a document (that is, an integer in the range $1, \ldots, n$). Hence, the precomputed samples require small additional storage. These sizes support the use of presampling (see Section 4).

5.2. Sparse nonnegative data

Our data here was a collection of 22,173 Reuters newswire articles. The matrix form of the data has 22,173 rows and 106,113 columns, with each column representing a word. Each matrix entry was a function of the word's frequency in the document, computed using the Cornell "ltc" weighting [5]. This weighting formula has been shown to be effective in making proximity of document vectors correspond more closely to similarity of meaning. The resulting matrix is sparse, with about 2×10^6 nonzeros. The vector for each document is normalized to have Euclidean norm of 1.0, so that the dot product between two document vectors is their cosine correlation.

A. Alg: H, T	CPU time (secs.) Index	Query	Presamples Computed	Samples Drawn	Dot Products Computed
Full 0,0	–	3708	0	0	25000000
R1 50,30	85	338	738801	450181950	657121
R2 50,25	85	634	738801	450181950	2625280
R3 60,30	86	629	886563	540218340	2284815
R4 60,25	86	1332	886563	540218340	6934072
R5 80,40	88	667	1182037	720291120	1879159

B.	Number of Pairs with Dot Product of at Least						
Alg: H, T	0.55	0.60	0.65	0.70	0.75	0.80	0.85
Full 0,0	106368	68872	44412	28320	17688	11256	7568
R1 50,30	90620	62216	41842	27403	17410	11184	7549
R2 50,25	103306	67784	44061	28217	17667	11256	7568
R3 60,30	104339	68223	44218	28259	17674	11255	7568
R4 60,25	106201	68829	44402	28320	17688	11256	7568
R5 80,40	105153	68554	44343	28305	17686	11256	7568

Table 1: Box B. shows number of pairs of similar LSI document vectors found by various sampling methods. For each method, Box A. gives both indexing time (time to build data structures prior to querying) and querying time (time to execute all queries), along with number of presamples drawn and samples used at query time (controlled by parameter H) and number of exact dot products computed (controlled by parameter T). Method *Full* is equivalent to using $H = 0$ and $T = 0$. It computes the dot product between all pairs of vectors, so counts listed for it are the total number of dot products at or over each threshold value.

Information retrieval systems typically compute the dot product of a query vector with a collection of sparse document vectors by making use of an inverted file [27] of document vectors. An inverted file is also the appropriate storage for efficient sparse matrix multiplication (see [16]). For each indexing term, a list of the id's of all documents with a nonzero weight for that term, plus the weights themselves, are stored. The dot product of a query vector with every document vector can be computed by examining only the inverted lists for those terms where the query vector has a nonzero value. We compared our sampling approach to computing the dot products with a publically available inverted file-based IR system, SMART[4].

5.2.1. Results

We experimented with the sampling algorithm by itself and in conjunction with dot product evaluation of high scoring matches. The use of sampling alone has the advantage of reduced storage requirements, since after preprocessing, only the (typically small) sample matrix is needed to resolve queries. The combined algorithm, however, yielded more accurate results.

In the previous section, we showed our combined algorithm to be highly effective for dense data, where eliminating even a portion of the expensive computations of explicit dot products gives a considerable speedup. For sparse data, however, sparse matrix multiplication techniques compute the full matrix product much faster than computing of each dot product individually would require. For the *Reuters* dataset, the time required for sparse multiplication was comparable to evaluating

[4]Available from directory */pub/smart* on *ftp.cs.cornell.edu*.

A. Alg: H,T	CPU time (secs.) Index	Query	Presamples Computed	Samples Drawn	Dot Products Computed
SMART	67	3503	–	–	491641929
Full	–	2266	–	–	491641929
R1 60,10	185	382	2266396	392812060	2043813
R2 60,7	185	529	2266396	392812060	5603726
R3 100,15	203	419	3307053	498441200	1317467
R4 100,11	203	517	3307053	498441200	3295974
R5 40,8	181	288	1526426	273542120	1486735

B.	Instances with Dot Product of at Least							
Alg: H,T	0.15	0.20	0.25	0.30	0.35	0.40	0.45	0.50
Full	1755221	550871	243193	142323	97035	71969	54921	43559
R1 60,10	0.641	0.885	0.975	0.995	0.999	1.000	1.000	1.000
R2 60,7	0.891	0.979	0.997	1.000	1.000	1.000	1.000	1.000
R3 100,15	0.480	0.749	0.906	0.970	0.992	0.998	0.999	1.000
R4 100,11	0.763	0.919	0.978	0.995	0.999	1.000	1.000	1.000
R5 40,8	0.494	0.763	0.918	0.976	0.994	0.998	0.999	1.000

Table 2: Box B. shows number of pairs of similar sparse document vectors found by various sampling methods. For each method, Box A. gives indexing and querying time, along with number of samples drawn and number of exact dot products computed. Both Method *Full* and the text retrieval system SMART use sparse matrix multiplication techniques to compute the dot products between all pairs of vectors. We show under *Full* in Box B. the total number of dot products at or over each threshold value. For the sampling algorithms, we instead show what proportion of those over-threshold dot products were found.

10% of the dot products individually. Hence, our combined algorithm is effective only if sampling can eliminate a very large fraction of the dot products that would normally need to be computed. This was indeed the case in our experiments for the *Reuters* dataset.

We tested the algorithm's performance with respect to the goals of identifying all k nearest neighbors and identifying all high matches. For these goals, sparse matrix multiplication achieves perfect results. The sampling based algorithms are faster but incur some loss of accuracy in the results. We examine these tradeoffs.

Indentifying all high similarities by a combined algorithm.

Table 2 compares the efficiency of sparse matrix multiplication with that of our combined algorithm when the goal is identifying and computing all dot products over a specified threshold. Two implementations of sparse matrix multiplication, our own (*Full*) and SMART, were used. SMART did not write the high dot products to a file, so its "Query" time in Table 2 slightly underestimates its time to complete the task. Also note that the time to build the inverted file is shown as "Index" time for SMART, but is included in "Query" time for *Full*. As in the previous experiment, H controls the size of the sampled representation used by our combined algorithm, while T controls how many instances have explicit dot products computed.

Table 2 shows that the combined algorithm uses substantially less time than either of the sparse matrix mutiplication implementations, while still finding a high proportion of the large dot products.

Identifying k nearest neighbors by a combined algorithm.

We also investigated identifying all k nearest neighbors using a combined algorithm with parameters H and k'. The sampling algorithm is applied first with parameter H. The k' highest scores are identified, and the exact dot products for these instances were computed. The algorithm returns these k' values sorted according to dot product value. The performance measure is the fraction of query documents whose k nearest neighbors are found in this set of k' items, for various $k \leq k'$. In the algorithms used here, we used the same parameter H for all queries. This means that we "lost" many nearest neighbors with low match values. We suspect that the performance can be improved considerably by determining H individually and online for each query: larger values of H are used when the nearest neighbors have low matches (see discussion in Sec. 4.4). In Table 3 we list results for several choices of H and k'. For each run, we list the index and query CPU times, the size of the sampled representation, the total number of samples used for queries, and the fraction of queries for which at least k Nearest Neighbors were correctly identified (for $k = 1, \ldots, 9$). The trivial Nearest Neighbor, the instance itself, is always excluded. Table 3 includes the running times of *Full* (our implementation) and *SMART* for computing all dot products and outputting the 10 nearest neighbors. The last two rows provide the performance for the ranked *union* of the lists of 10 top documents from two experiments. We listed the results for the union of R2 and R4 and the union of R5 and R6.

Algorithm R5+R6 used 38% of the query processing time of *full* and found 99% of the nearest neighbors and 99% of the 4th nearest neighbors. Algorithm R4 used 19% of the time and found 90% of nearest neighbors and 50% of the 4th nearest neighbors. We observed that when k nearest neighbors are not identified correctly, it is often because the kth nearest neighbor has a low match and is close in value to a large set of other instances. In most applications it would not be important to distinguish the kth nearest neighbor from other instances with similar dot products.

Identifying nearest neighbors by sampling only.

In addition to the above experiments with the combined algorithm, we used a pure sampling algorithm to find the single nearest neighbor for each of the $22,173$ *Reuters* document vector. (The trivial nearest neighbor, the instance itself, was excluded.) Two pure sampling runs were done, one taking 337,826,300 samples (Index time of 141 seconds and Query time of 299 seconds), and one taking 136,777,240 samples (Index time of 135 seconds and Query time of 140 seconds). For the first run we used hit factor $H = 50$ and it required 1902250 presamples. For the second we used $H = 20$ and it required 768923 presamples. The times here are similar or better than those of the combined algorithm, and like the combined algorithm much better than those of both *Full* and SMART. However, the accuracy of the pure sampling runs was considerably inferior to that of the pure algorithm. The slower sampling run correctly found the first nearest neighbor of a document 9,352 times (a proportion of 0.422), while the faster run found the first nearest neighbor only 6,311 times (0.285). The nearest neighbor was in the top 10 documents (breaking ties arbitrarily) 16,596 times (0.748) for the first run, and 12,558 times (0.566) in the second. It should be noted that in many cases the nearest neighbor is one of a substantial number of documents with very close similarities to the query document, and the above statistics penalize the sampling method for finding those very similar documents. Nevertheless, the pure sampling method seems considerably inferior to the combined one.

A. Alg: H, k'	CPU time (secs.) Index	Query	Presamples Computed	Samples Drawn
SMART	67	3628	0	0
Full	–	2591	0	0
R1 30,40	177	240	1147270	205282680
R4 40,50	179	318	1526426	273542120
R2 50,50	182	374	1902252	337826300
R5 50,100	182	454	1902252	337826300
R6 100,100	183	516	3307053	498441200
R8 140,50	190	462	3816159	517037360
R2 + R4	–	690	3428678	611368420
R5 + R6	–	972	5209305	836267500

B.	Fraction of queries where all k Nearest Neighbors are Found								
Alg: H, k'	1NN	2NN	3NN	4NN	5NN	6NN	7NN	8NN	9NN
SMART	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
Full	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
R1 30,40	0.836	0.642	0.478	0.355	0.264	0.200	0.154	0.119	0.030
R4 40,50	0.901	0.764	0.625	0.507	0.409	0.332	0.271	0.221	0.181
R2 50,50	0.929	0.817	0.700	0.593	0.492	0.410	0.343	0.288	0.240
R5 50,100	0.956	0.881	0.795	0.709	0.624	0.548	0.481	0.423	0.371
R6 100,100	0.970	0.916	0.854	0.785	0.720	0.659	0.599	0.546	0.498
R8 140,50	0.956	0.886	0.805	0.725	0.649	0.580	0.518	0.458	0.402
R2 + R4	0.972	0.922	0.859	0.793	0.722	0.657	0.592	0.531	0.479
R5 + R6	0.992	0.976	0.951	0.922	0.889	0.851	0.813	0.775	0.738

Table 3: Proportion of the k nearest neighbors identified for *Reuters* data, for various values of k and sampling algorithms.

6. k-matrix products in text retrieval

Our implementation and experiments were performed for applications where the database is a single matrix $\mathbf{A}$. Text retrieval applications exist where the database is represented as a product of several matrices, and our implementation ideas extend to this case (see Sec. 4.2). In these applications, retrieval from a sparse document-term matrix is enhanced by using a manually or automatically generated thesaurus [18, 23]. The original database matrix $\mathbf{A}^{n \times d}$ is a sparse document by term matrix. Each query vector $\mathbf{q}$ has dimension d (number of terms). The thesaurus is a sparse d by d matrix $\mathbf{T}$. The vector $\mathbf{q}^T\mathbf{T}$ semantically corresponds to smoothing $\mathbf{q}$ by the thesaurus. More meaningful retrieval results can sometimes be obtained from the enhanced products $\mathbf{q}^T\mathbf{T}\mathbf{A}^T$ or $\mathbf{q}^T\mathbf{T}\mathbf{T}^T\mathbf{A}^T$ (rather than $\mathbf{q}^T\mathbf{A}^T$). It is generally not effective, storage or time-wise, to explicitly compute $\mathbf{P} = \mathbf{T}\mathbf{A}^T$ and use the single-matrix implementation. Computing $\mathbf{P}$ could be computationally extensive and moreover, $\mathbf{P}$ is generally much denser than its product form. Hence, the use of a k-matrix implementation would be an effective approach.

7. Discussion

We have shown how a sampling-based method for approximating matrix multiplication can increase the efficiency of dot product computations for pattern recognition. On dense (i.e. non-sparse) data sets, our approach is particularly effective. While there has been considerable progress in speeding up the *production* of LSI document representations [4], there has been little progress in speeding up *retrieval* with LSI document representations, beyond the obvious expedients of using parallel hardware or using fewer latent dimensions [14]. To achieve our speedup of 5 to 10-fold by dropping dimensions would require using only 32–64 dimensions, which would substantially impact effectiveness.

Our results on sparse data should be evaluated cautiously, since a wide variety of methods for speeding up inverted file based scoring are already known, particularly when only the top scoring instances are needed and exact results are not necessary [25, 27]. These methods include compressing inverted files (to reduce I/O), using lower precision arithmetic, ignoring some inverted lists or parts of inverted lists, and limiting the number of document score accumulators maintained. Our 5 to 15-fold speedup over inverted file based matching therefore must be considered in the context of these alternative optimizations. Our method does have advantage that the expected values of our scores are equal to the true scores and the variance of our estimates is predictable and well behaved.

Acknowledgement. The authors are thankful to Yoav Freund for stimulating discussions and helpful comments.

References.

[1] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Wu. An optimal algorithm for approximate nearest neighbor searching. In *Proc. 5th ACM-SIAM Symposium on Discrete Algorithms*, pages 573–582. ACM-SIAM, 1994.

[2] K. E. Atkinson. *Numerical Analysis.* John Wiley & Sons, New York, second edition, 1989.

[3] J. L. Bentley, B. W. Weide, and A. C. Yao. Optimal expected-time algorithms for closest point problems. *ACM Trans. Math. Software*, 6:563–580, 1980.

[4] M. W. Berry, S. T. Dumais, G. W. O'Brien. Using linear algebra for intelligent information retrieval. *SIAM Review*, 37(4): 573–595, 1995.

[5] C. Buckley, G. Salton, and J. Allan. The effect of adding relevance information in a relevance feedback environment. In W. B. Croft and C. J. van Rijsbergen, editors, *SIGIR 94: Proceedings of the Seventeenth Annual International ACM-SIGIR Conference on Research and Development in Information Retrieval*, pages 292–300, London, 1994. Springer-Verlag.

[6] H. Chernoff. A measure of the asymptotic efficiency for test of a hypothesis based on the sum of observations. *Annals of Math. Statistics*, 23:493–509, 1952.

[7] K. L. Clarkson. Personal communication, December 1996.

[8] K. L. Clarkson. Rapid estimation of matrix-vector products and applications. Manuscript, October 96.

[9] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. In *Proc. 19th Annual ACM Symposium on Theory of Computing*, pages 1–6. ACM, 1987.

[10] W. B. Croft and D. J. Harper. Using probabilistic models of document retrieval without relevance feedback. *Journal of Documentation*, 35(4):285–295, 1979.

[11] B. V. Dasarathy, editor. *Nearest Neighbor (NN) Norms: NN Pattern Classification Techniques.* IEEE Computer Society Press, Los Alamitos, CA, 1991.

[12] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman. Indexing by latent semantic indexing. *Journal of the American Society for Information Science*, 41(6):391–407, September 1990.

[13] R. O. Duda and P. E. Hart. *Pattern Classification and Scene Analysis.* Wiley-Interscience, New York, 1973.

[14] S. T. Dumais. Latent semantic indexing (LSI): TREC-3 report. In D. K. Harman, editor, *The Third Text Retrieval Conference (TREC-3)*, pages 219–230, Gaithersburg, MD, 1995. U. S. Dept. of Commerce, National Institute of Standards and Technology.

[15] W. Feller. *An introduction to probability theory and its applications*, volume 1. John Wiley & Sons, New York, 1968.

[16] G. Golub. *Matrix Computations.* The Johns Hopkins U. Press, Baltimore, MD, 1989.

[17] T. J. Hastie and R. J. Tibshirani. *Generalized Additive Models.* Chapman & Hall, London, 1990.

[18] D. D. Lewis. An evaluation of phrasal and clustered representations on a text categorization task. In N. Belkin, P. Ingwersen, and A. M. Pejtersen, editors, *SIGIR '92: Proceedings of the Fifteenth Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 37–50, 1992.

[19] D. D. Lewis, R. E. Schapire, J. P. Callan, and R. Papka. Training algorithms for linear text classifiers. In H. P. Frei, et al, editors, *SIGIR '96: Proceedings of the 19th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 298–306, 1996.

[20] D. Michie, D. J. Spiegelhalter, and C. C. Taylor, editors. *Machine Learning, Neural and Statistical Classification.* Ellis Horwood, New York, 1994.

[21] S. E. Robertson and K. Sparck Jones. Relevance weighting of search terms. *Journal of the American Society for Information Science*, pages 129–146, May-June 1976.

[22] G. Salton and M. J. McGill. *Introduction to Modern Information Retrieval.* McGraw-Hill Book Company, New York, 1983.

[23] K. Sparck Jones. *Automatic Keyword Classification for Information Retrieval.* Archon Books, 1971.

[24] R. H. Thompson and W. B. Croft. Support for browsing in an intelligent text retrieval system. *International Journal of Man-Machine Studies*, 30:639–668, 1989.

[25] H. Turtle and J. Flood. Query evaluation: Strategies and optimizations. *Information Processing and Management*, 31(6):831–850, 1995.

[26] P. Willett. Recent trends in hierarchic document clustering: A critical review. *Information Processing and Management*, 24(5):577–598, 1988.

[27] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images.* Von Nostrand Reinhold, New York, 1994.

[28] Y. Yang and C. G. Chute. An example-based mapping method for text categorization and retrieval. *ACM Transactions on Information Systems*, 12(3):252–277, July 1994.