Predictive Indexing for Fast Search

Sharad Goel, John Langford and Alex Strehl

Yahoo! Research, New York

Modern Massive Data Sets (MMDS)
June 25, 2008

Large-Scale Search

Large-scale search is now a ubiquitous operation:

- Web Search — 10 billion web pages; given a search query, have 10ms to return top web page results.
- Web Advertising — 1 billion ads; given a web page, have 10ms to choose best ads for the page.
- Nearest Neighbor — N points in Euclidean space; given a query point, quickly return the nearest k neighbors. This problem arises in data compression, information retrieval, and pattern recognition.

Large-scale search algorithms must satisfy tight computational restrictions. In practice, we are usually satisfied with approximate solutions.

Large-Scale Search — Current Approaches

Existing large-scale search solutions often work by preparing the query string and repeatedly filtering the dataset.

In web search, given the query string "the rain in Spain stays mainly in the plain," one approach is to

1. Rewrite the query string:

   "~~the~~ (rain *or* rains) ~~in~~ Spain stays mainly ~~in the~~ (plain *or* plains)"
2. Retrieve web pages containing all the query terms
3. Filter these web pages by, for example, PageRank
4. Filter some more...
5. Return top 10 web page results

Large-Scale Search — Current Approaches

In this approach of query rewriting and successive filtering, a result p is "good" for a query q if the search system generates that result in response to the query. That is, fitness is determined by architecture.

Alternatively, we could a priori define how "good" a page is for a query, and then build a system to support that definition. In this way, architecture is determined by fitness.

For this latter approach, we need two ingredients:

1. A (machine-learned) score $f(q, p)$ defined for every query/page pair
2. A method for quickly returning the highest scoring pages for any query

Here we consider only the second problem, rapid retrieval.

The Problem

Consider

- $Q \subseteq \mathbb{R}^n$ an *input space*
- $W \subseteq \mathbb{R}^m$ a finite *output space* of size N
- $f : Q \times W \mapsto \mathbb{R}$ a known *scoring function*.

Given an input (search query) $q \in Q$, the goal is to find, or closely approximate, the top-k output objects (web pages) $p_1, \ldots, p_k$ in W (i.e., the top k objects as ranked by $f(q, \cdot)$).

Feature Representation

One concrete way to map web search into this general framework is to represent both queries and pages as sparse binary feature vectors in a high-dimensional Euclidean space.

Specifically, we associate each word with a coordinate: A query (page) has a value of 1 for that coordinate if it contains the word, and a value of 0 otherwise. We call this the word-based feature representation, because each query and page can be summarized by a list of its features (i.e., words) that it contains.

The general framework supports many other possible representations, including those that incorporate the difference between words in the title and words in the body of the web page, the number of times a word occurs, or the IP address of the user entering the query.

Related Work

A standard approach to the rapid retrieval problem is to pre-compute an index (or other datastructure) in order to significantly reduce runtime computation.

There are two common techniques for doing this:

- Fagin's Threshold Algorithm
  We use this as a baseline comparison.
- Inverted Indices
  A datastructure that maps every page feature (i.e., word) to a list of pages that contain that feature. This works best for "sparse" scoring rules, which is not typical for the machine learned rules we consider.

Predictive Indexing — Summary

Suppose we are provided with a categorization of possible queries into related, potentially overlapping, sets. For example, one set might be defined as, "queries containing the word 'Spain'".

We assume queries are generated from some fixed distribution.

For each query set, the associated predictive index is an ordered list of web pages sorted by their expected score for random queries drawn from that set.

In particular, we expect web pages at the top of the 'Spain' list to be good, on average, for queries containing the word 'Spain' In contrast to an inverted index, the pages in the 'Spain' list need not themselves contain the word 'Spain'.

To retrieve results for a particular query, we optimize only over web pages in the relevant, pre-computed lists.

Predictive Indexing — Formal Description

Consider a finite collection $\mathcal{Q}$ of sets $Q_i \subseteq Q$ that cover the query space (i.e., $Q \subseteq \cup_i Q_i$).

For each Q_i, define the conditional probability distribution D_i over queries in Q_i by $D_i(\cdot) = D(\cdot|Q_i)$.

Define $f_i : W \mapsto \mathbb{R}$ as $f_i(p) = \mathbb{E}_{q \sim D_i}[f(q, p)]$.

The function $f_i(p)$ is the expected score of the web page p for the (related) queries in Q_i.

Predictive Indexing — Formal Description

For each query set we pre-compute a sorted list of web pages ordered by $f_i(p)$.

At runtime, we search down the lists that correspond to query sets containing the given query, halting when we have exhausted the computational budget.

the	*rain*	*in*	*Spain*	...
p_{53}	p_6	p_{96}	p_{58}	⋮
p_{64}	p_{91}	p_{58}	p_{21}	⋮
p_{39}	p_1	p_{65}	p_{43}	⋮
⋮	⋮	⋮	⋮	⋮
p_{76}	p_7	p_{58}	p_{42}	⋮

$$f_{rain}(p_6) \geq f_{rain}(p_{91}) \geq f_{rain}(p_1) \geq \cdots \geq f_{rain}(p_7) > 0.$$

Predictive Indexing — Formal Description

To generate the predictive index, we do not assume anything about the particular structure of the scoring function.

We do assume, however, that we can sample from the query distribution, in order to approximate the conditional expected scores.

for t random queries $q \sim D$ **do**
  for all query sets Q_j containing q **do**
    for all pages p in the data set **do**
      $L_j[p] \leftarrow L_j[p] + f(q, p)$
    end for
  end for
end for

for all lists L_j **do**
  sort L_j
end for

Empirical Evaluation

We evaluate predictive indexing for two applications:

- Internet advertising
- Approximate nearest neighbor

Empirical Evaluation — Internet Advertising

We consider commercial web advertising data. The data are comprised of logs of events, where each event represents a visit by a user to a particular web page p, from a set of web pages $Q \subseteq \mathbb{R}^n$ (we use a sparse feature representation).

From a large set of advertisements $W \subseteq \mathbb{R}^m$, the commercial system chooses a smaller, ordered set of ads to display on the page (generally around 4). The set of ads seen and clicked by users is logged.

Note that the role played by web pages has switched, from result to query.

- The training data consist of 5 million events (web page $\times$ ad displays)
- About 650,000 distinct ads
- Each ad contains, on average, 30 ad features (out of $\approx 1,000,000$)
- About 500,000 distinct web pages
- Each page consists of approximately 50 page feature (out of $\approx 900,000$)

Empirical Evaluation — Internet Advertisting

We trained a linear scoring rule f of the form

$$f(p, a) = \sum_{i,j} w_{ij} p_i a_j$$

to approximately rank the ads by their probability of click. Here, w_{ij} are the learned weights (parameters) of the linear model, and $p_i, a_j \in \{0, 1\}$.

We measured computation time in terms of the number of full evaluations by the algorithm (i.e., the number of ads scored against a given page).

Thus, the true test of an algorithm was to quickly select the most promising $T \in \{100, 200, 300, 400, 500\}$ ads (out of 650,000) to fully score against the page.

Empirical Evaluation — Internet Advertising

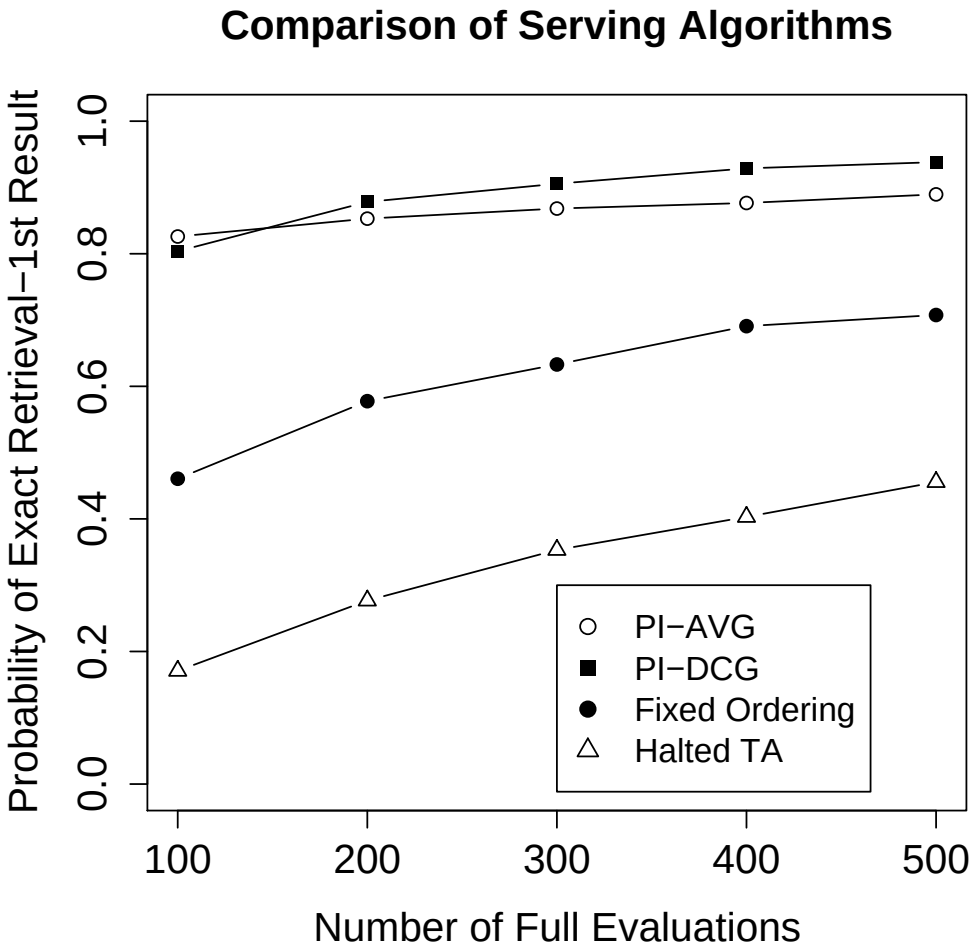
We tested four methods:

- Halted Threshold Algorithm (TA)
- Predictive Indexing
  Each feature corresponds to a cover set, namely the set of queries that contain that feature
  - PI-AVG — pages ordered by expected score
  - PI-DCG — pages ordered by probability of being a top result

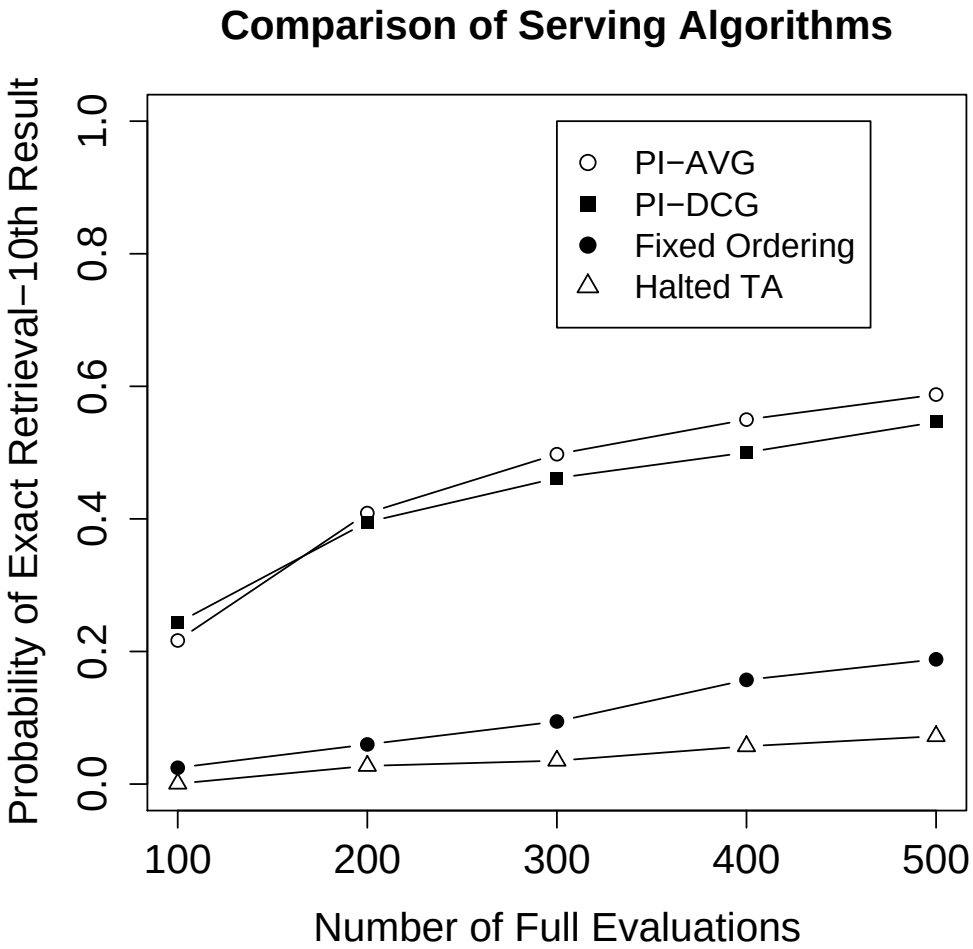
$$\mathrm{DCG}_f(p, a) = I_{r(p,a)\leq 16} / \log_2 (r(p, a) + 1).$$

- Best global ordering (BO)
  A degenerate form of predictive indexing that uses a single cover set.
  Pages ordered by $\mathrm{DCG}_f(p, a)$.

Empirical Evaluation — Internet Advertising



Empirical Evaluation — Internet Advertising



Empirical Evaluation — Approximate Nearest Neighbor

A special case application of predictive indexing is approximate nearest neighbor search.

Given a set of points in n-dimensional Euclidean space, and a query point x in that same space, the nearest neighbor problem is to quickly return the top-k neighbors of x.

This problem is of considerable interest for a variety of applications, including data compression, information retrieval, and pattern recognition.

In the predictive indexing framework, the nearest neighbor problem corresponds to minimizing a scoring function defined by Euclidean distance:

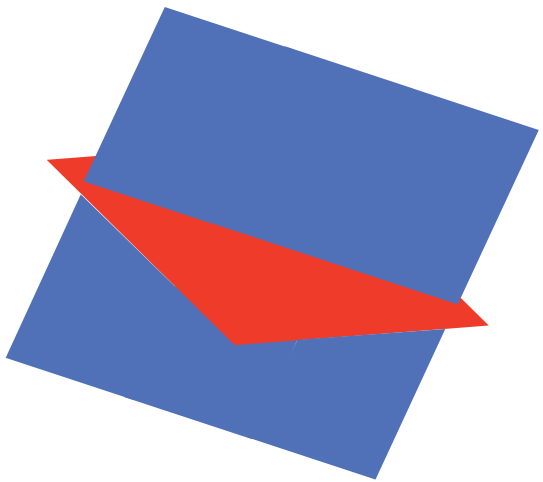
$$f(x, y) = ||x - y||_2$$

Empirical Evaluation — Approximate Nearest Neighbor

To start, we define a covering of the input space $\mathbb{R}^n$, which we borrow from locality-sensitive hashing (LSH), a commonly suggested scheme for the approximate nearest neighbor problem.

We form α random partitions of the input space.

Each partition splits the space by β random hyperplanes.



In total, we have $\alpha \cdot 2^{\beta}$ cover sets, and each point is covered by α sets.

Empirical Evaluation — Approximate Nearest Neighbor

Given a query point x, LSH evaluates points in the α sets that cover x.

Predictive indexing, in contrast, maps each cover set C to an ordered list of points sorted by their probability of being a top-10 nearest point to points in C.

That is, the lists are sorted by

$$h_C(p) = \Pr_{q \sim D|C}(p \text{ is one of the nearest 10 points to } q).$$

For the query x, we then consider those points with large probability h_C for at least one of the α sets that cover x.

Empirical Evaluation — Approximate Nearest Neighbor

We compare LSH and predictive indexing over four public data sets:

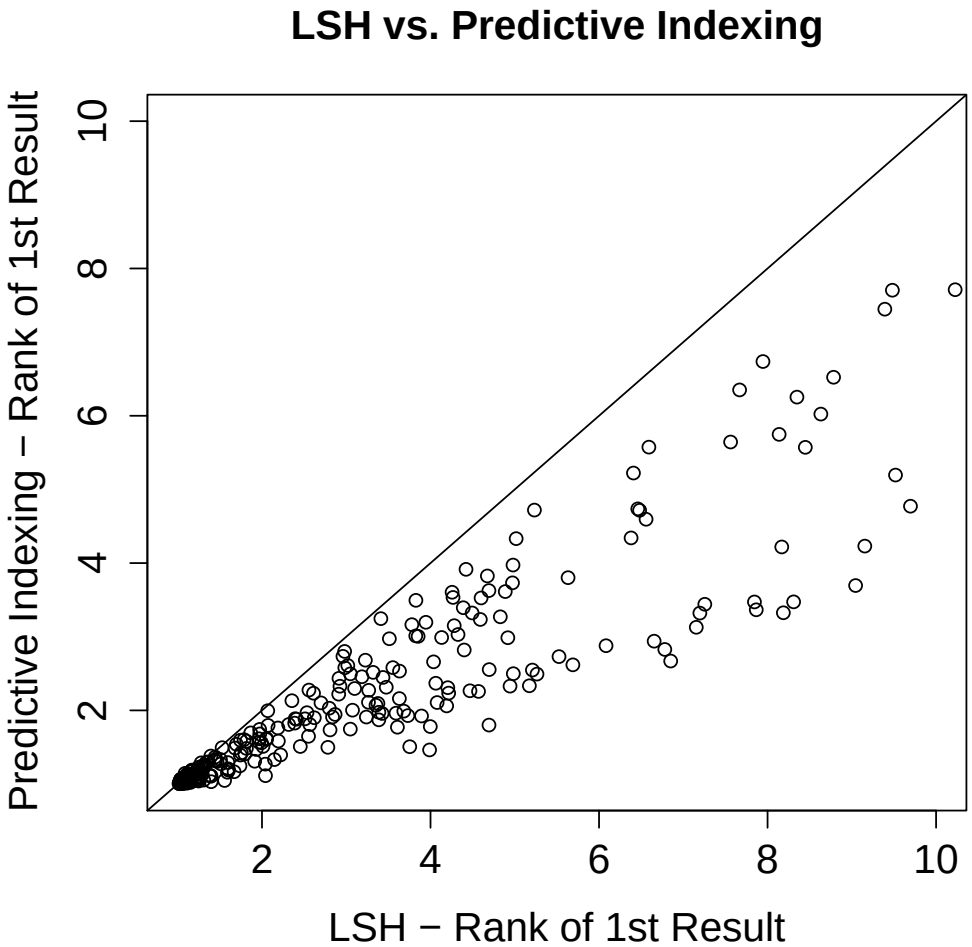
1. MNIST — 60,000 training and 10,000 test points in 784 dimensions ($\beta = 24$ projections per partition)
2. Corel — 37,749 points in 32 dimensions, split randomly into 95% training and 5% test subsets ($\beta = 24$ projections per partition)
3. Pendigits — 7494 training and 3498 test points in 17 dimensions ($\beta = 63$ projections per partition)
4. Optdigits — 3823 training and 1797 test points in 65 dimensions ($\beta = 63$ projections per partition)

The number of partitions α was varied as an experimental parameter.

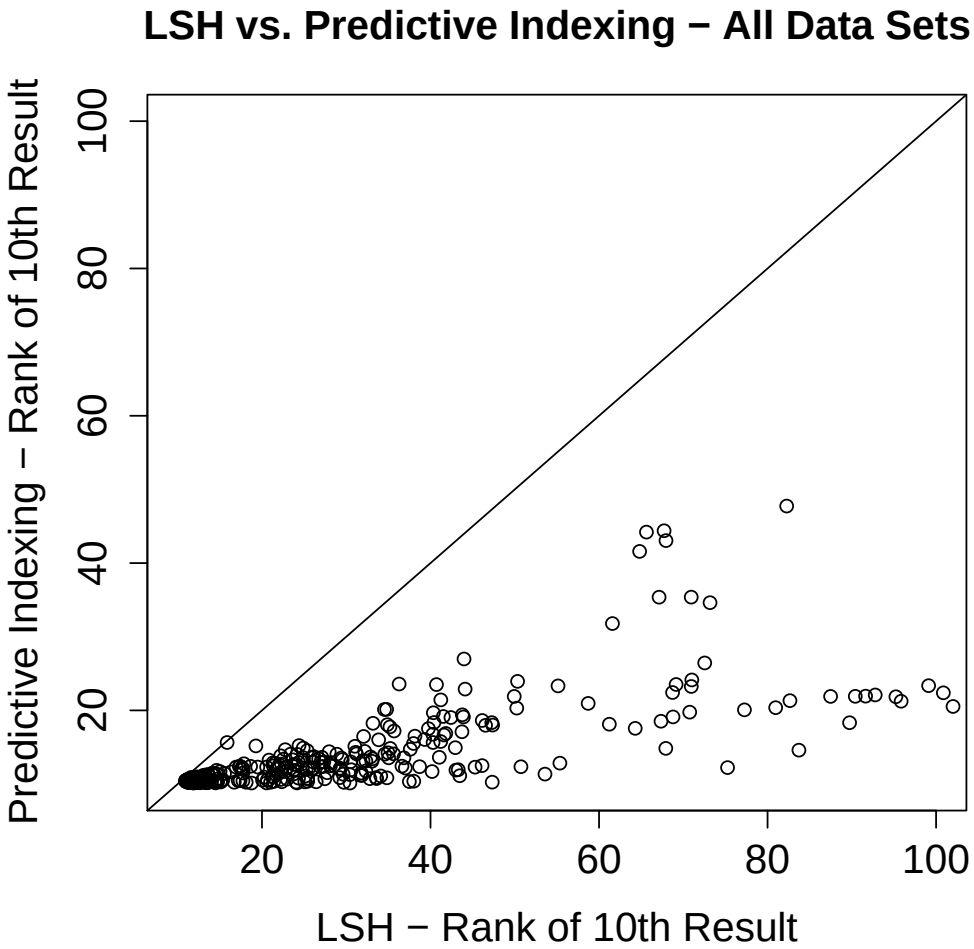
Larger α corresponds to more full evaluations per query, resulting in improved accuracy at the expense of increased computation time.

Both algorithms allowed the same average number of full evaluations per query.

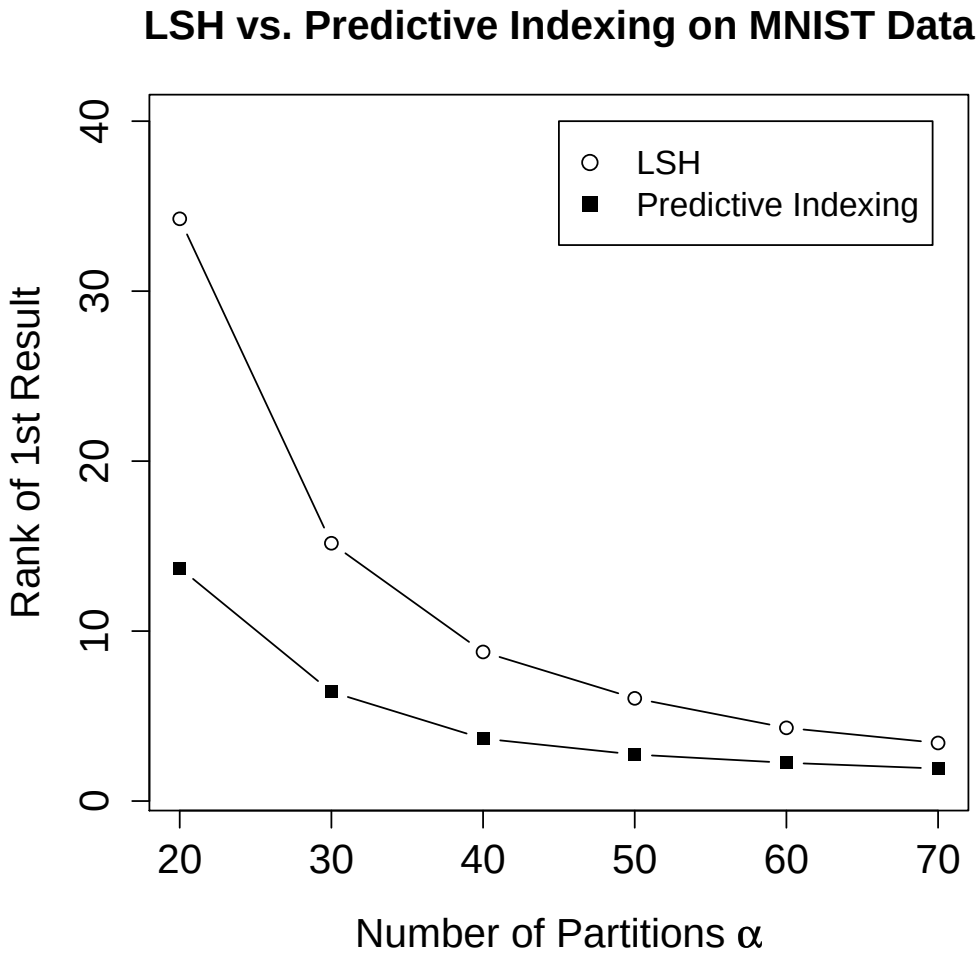
Empirical Evaluation — Approximate Nearest Neighbor



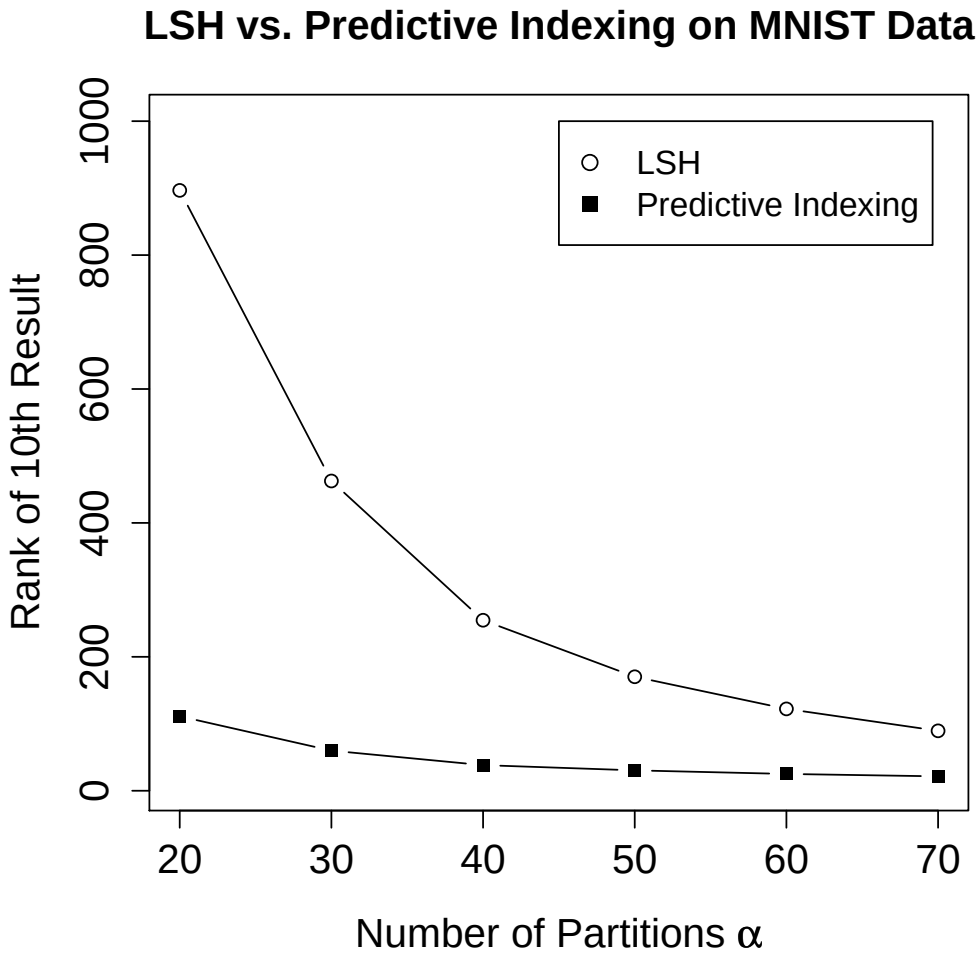
Empirical Evaluation — Approximate Nearest Neighbor



Empirical Evaluation — Approximate Nearest Neighbor



Empirical Evaluation — Approximate Nearest Neighbor



Conclusion

Predictive indexing is based on a very simple idea: Build an index that incorporates the query distribution.

The method performs well in our empirical evaluations of Internet advertising display and the approximate nearest neighbor problem.

We believe the predictive index is the first datastructure capable of supporting scalable rapid ranking based on general purpose machine-learned scoring rules.

Appendix

Related Work — Fagin's Threshold Algorithm

Consider the word-based feature representation, and suppose the scoring function has the form

$$f(q,p) = \sum_{\{i:q_i=1\}} g_i(p).$$

For example, the query "the rain in Spain" corresponds to

$$f(q,p) = g_{the}(p) + g_{rain}(p) + g_{in}(p) + g_{Spain}(p)$$

where $g_i(p)$ is some measure of how important term i is to document p (e.g., based on tf-idf)

Related Work — Fagin's Threshold Algorithm

For each query feature (i.e., word), TA pre-computes an ordered list L_i sorted by $g_i(p)$.

the	*rain*	*in*	*Spain*	...
p_{53}	p_6	p_{96}	p_{58}	⋮
p_{64}	p_{91}	p_{58}	p_{21}	⋮
p_{39}	p_1	p_{65}	p_{43}	⋮
⋮	⋮	⋮	⋮	⋮
p_{76}	p_7	p_{58}	p_{42}	⋮

$$g_{rain}(p_6) \geq g_{rain}(p_{91}) \geq g_{rain}(p_1) \geq \cdots \geq g_{rain}(p_7) > 0.$$

Related Work — Fagin's Threshold Algorithm

Given a query (e.g., "the rain in Spain"), TA searches down the relevant lists (e.g., L_{the}, L_{rain}, L_{in}, and L_{Spain}) in parallel, fully evaluating each new page that it comes across.

Clearly, any page with non-zero score must be in one of these lists. So, searching over these lists is sufficient.

TA, however, avoids searching over all the pages in these lists by maintaining upper and lower bounds on the score of the k^{th} best page, halting when these bounds cross.

The lower bound is the score of the k^{th} best page evaluated so far.

Since the lists are ordered by their partial scores $g_i(p)$, the sum of the partial scores for next-to-be-scored page in each list bounds the score of any page yet to be seen, giving an upper bound.

Related Work — Fagin's Threshold Algorithm

Query = "the rain in Spain"

Upper Bound: $+\infty$
Lower Bound: $+\infty$

the	*rain*	*in*	*Spain*
p_{53}	p_6	p_{96}	p_{58}
p_{64}	p_{91}	p_{58}	p_{21}
p_{39}	p_1	p_{65}	p_{43}
$\vdots$	$\vdots$	$\vdots$	$\vdots$
p_{76}	p_7	p_{58}	p_{42}

Related Work — Fagin's Threshold Algorithm

Query = "the rain in Spain"

Upper Bound: 93.23
Lower Bound: 26.42

the	*rain*	*in*	*Spain*
p_{53}	p_6	p_{96}	p_{58}
p_{64}	p_{91}	p_{58}	p_{21}
p_{39}	p_1	p_{65}	p_{43}
⋮	⋮	⋮	⋮
p_{76}	p_7	p_{58}	p_{42}

Related Work — Fagin's Threshold Algorithm

Query = "the rain in Spain"

Upper Bound: 87.17
Lower Bound: 80.31

the	*rain*	*in*	*Spain*
p_{53}	p_6	p_{96}	p_{58}
p_{64}	p_{91}	p_{58}	p_{21}
p_{39}	p_1	p_{65}	p_{43}
⋮	⋮	⋮	⋮
p_{76}	p_7	p_{58}	p_{42}

Related Work — Fagin's Threshold Algorithm

Query = "the rain in Spain"

Upper Bound:	85.92
Lower Bound:	85.92

the	*rain*	*in*	*Spain*
p_{53}	p_6	p_{96}	p_{58}
p_{64}	p_{91}	p_{58}	p_{21}
p_{39}	p_1	p_{65}	p_{43}
⋮	⋮	⋮	⋮
p_{76}	p_7	p_{58}	p_{42}

Related Work — Fagin's Threshold Algorithm

The Fagin's Threshold Algorithm is particularly effective when a query contains a small number of features, facilitating fast convergence of the upper bound.

In our experiments, however, we find that the halting condition is rarely satisfied within the imposed computational restrictions.

One can, of course, simply halt the algorithm when it has expended the computational budget, which we refer to as the Halted Threshold Algorithm. We use this method as a baseline comparison.

Related Work — Inverted Index

An inverted index is a datastructure that maps every page feature i (i.e., word) to a list of pages p that contain i.

When a new query arrives, a subset of page features relevant to the query is first determined. For instance, if the query contains "rain", the page feature set might be {"rain", "precipitation", "drizzle", ...}.

A distinction is made between query features and page features, and in particular, the relevant page features may include many more words than the query itself.

Once a set of page features is determined, their respective lists (i.e., inverted indices) are searched, and from them the final list of output pages is chosen.

Related Work — Inverted Index

Approaches based on inverted indices are efficient only when it is sufficient to search over a relatively small set of inverted indices for each query.

Inverted indices require, for each query q, that there exists a small set of page features (typically ≤ 100) such that the score of any page against q depends only on those features.

In other words, the scoring rule must be extremely sparse, with most words or features in the page having zero contribution to the score for q.

We consider a machine-learned scoring rule, derived from internet advertising data, with the property that almost all page features have substantial influence on the score for every query, making any straightforward approach based on inverted indices intractable.