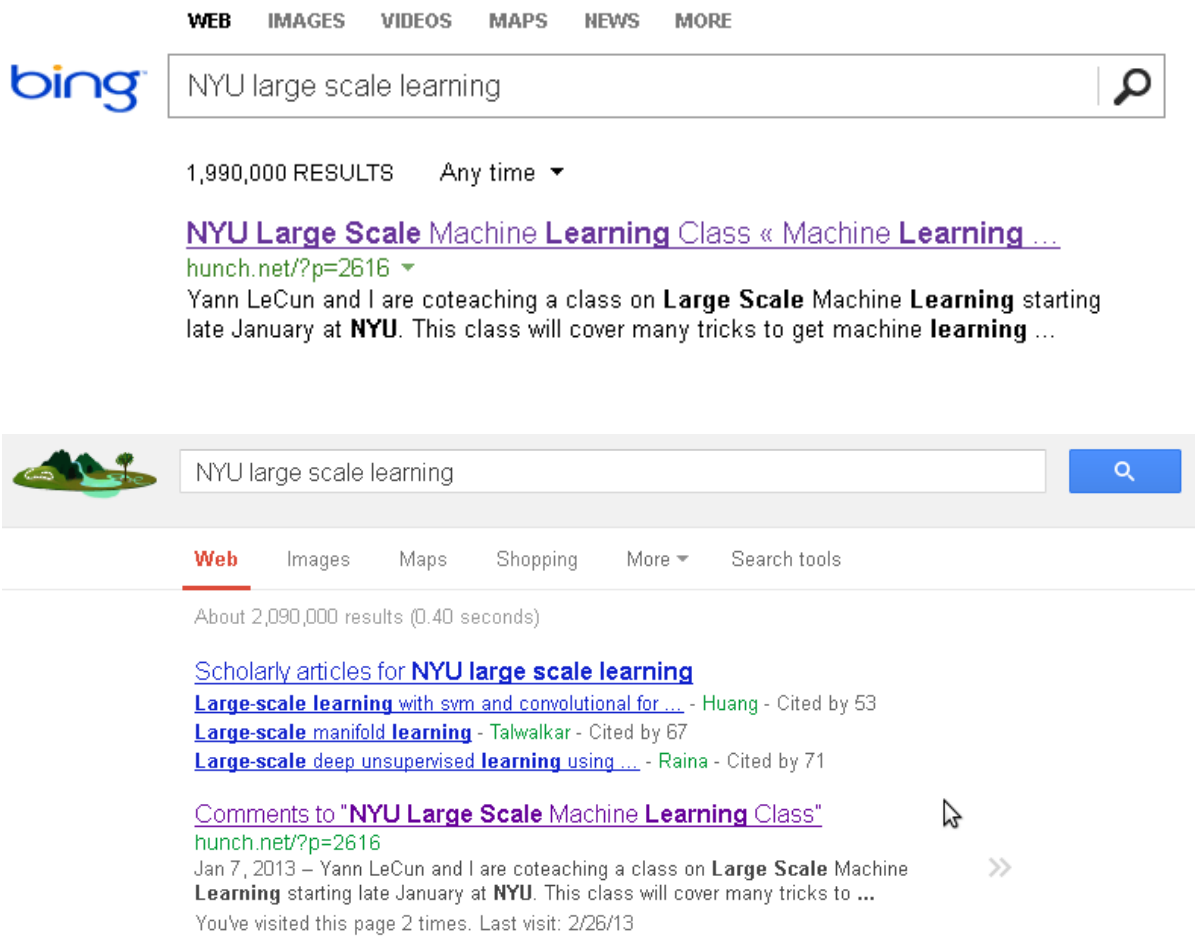
Indexing and Machine Learning

John Langford @ Microsoft Research

NYU Large Scale Learning Class, April 23

A Scenario

You have 10^{10} webpages and want to return the best result in 100ms.



How do you do it?

Method 1: Linear Scan

"Best" is defined by some (learned) quality score $s(q, r)$ where q is the query and r is the result.
Linear scan computes $\arg\max_r s(q, r)$ in linear time.

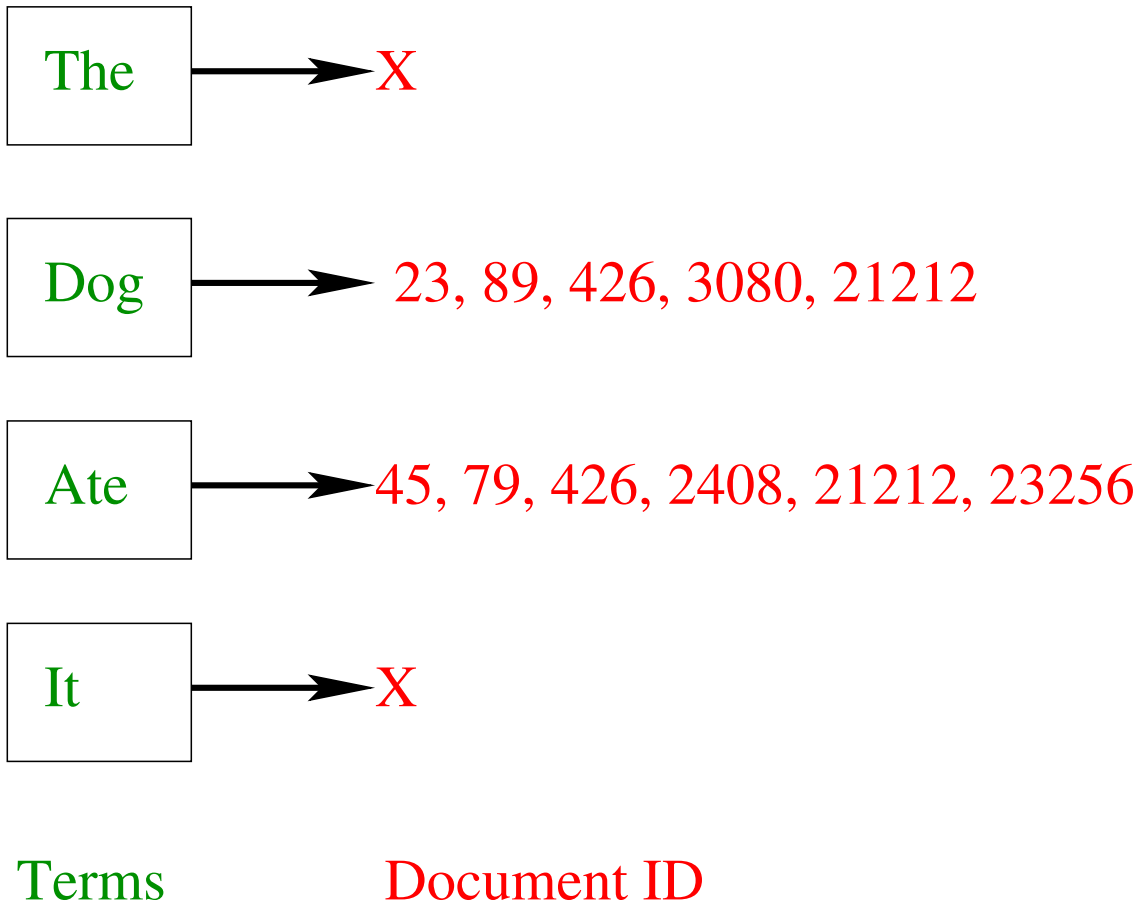
Method 1: Linear Scan

"Best" is defined by some (learned) quality score $s(q, r)$ where q is the query and r is the result.
Linear scan computes $\arg\max_r s(q, r)$ in linear time.

Need perhaps $10^{13}(?)$ cores. Luckily, there are other approaches.

Method 2: Inverted index

Inverted index = lookup table of documents containing a word.
[variants]



"Stop words" are unindexed (index too large).

Inverted Index Ops

What is efficient?

Inverted Index Ops

What is efficient? Set queries.

Inverted Index Ops

What is efficient? Set queries.

Use same sort over documents $\Rightarrow$ intersection of sets.

Union is inherently slower but possible by excluding sufficient stop words.

Inverted Index Ops

What is efficient? Set queries.

Use same sort over documents $\Rightarrow$ intersection of sets.

Union is inherently slower but possible by excluding sufficient stop words.

Problem remaining: $s(q, r)$ isn't a simple boolean of sets.
Induced Machine Learning Problem: How do you reformat/canonicalize queries so they pull up the right results?

Method 3: Weighted AND (WAND)

A generalization of an inverted index.

WAND $= \sum_i w_i I_i \geq \theta$ where $w_i > 0, \theta > 0$ and $I_i = 1$ if a term is present and 0 otherwise.

A WAND query can be evaluated efficiently by a clever algorithm using upper bounds and monotonicity.

The ML perspective: Closer to a learned rule, but still quite limited.

Method 4: Locality Sensitive Hashing

Precompute b random vectors $z_1, ..., z_b$
Represent each item with a vector x.
Compute a b-bit hash for each item where bit i satisfies $h_i(x) = I(x \cdot z_i > 0)$.
Store x in a lookup table indexed by $h(x)$.

Method 4: Locality Sensitive Hashing

Precompute b random vectors $z_1, ..., z_b$
Represent each item with a vector x.
Compute a b-bit hash for each item where bit i satisfies $h_i(x) = I(x \cdot z_i > 0)$.
Store x in a lookup table indexed by $h(x)$.

When a query q comes in compute hash and lookup matching x in table.

Method 4: Locality Sensitive Hashing

Precompute b random vectors $z_1, ..., z_b$
Represent each item with a vector x.
Compute a b-bit hash for each item where bit i satisfies $h_i(x) = I(x \cdot z_i > 0)$.
Store x in a lookup table indexed by $h(x)$.

When a query q comes in compute hash and lookup matching x in table.

Theorem: For sufficiently large h the closest match is returned with high probability (over the random projection).

Induced Machine Learning Problem: How do you map query and answer into the same space?

Method 4: Locality Sensitive Hashing

Precompute b random vectors $z_1, ..., z_b$
Represent each item with a vector x.
Compute a b-bit hash for each item where bit i satisfies $h_i(x) = I(x \cdot z_i > 0)$. [Variants]
Store x in a lookup table indexed by $h(x)$.

When a query q comes in compute hash and lookup matching x in table. [Variants]

Theorem: For sufficiently large h the closest match is returned with high probability (over the random projection).[Variants]

Induced Machine Learning Problem: How do you map query and answer into the same space?

Method 5: Predictive Indexing

For every word/key i construct a sorted list where the list is sorted according to $E[s(q, r)|i \in q]$ or $P(r \texttt{ best}|i \in q]$.

Method 5: Predictive Indexing

For every word/key i construct a sorted list where the list is sorted according to $E[s(q, r)|i \in q]$ or $P(r \texttt{ best}|i \in q]$.

To query, do a breadth first traversal over lists associated with each $i \in q$ doing a full evaluation. When time runs out, return the best result seen.

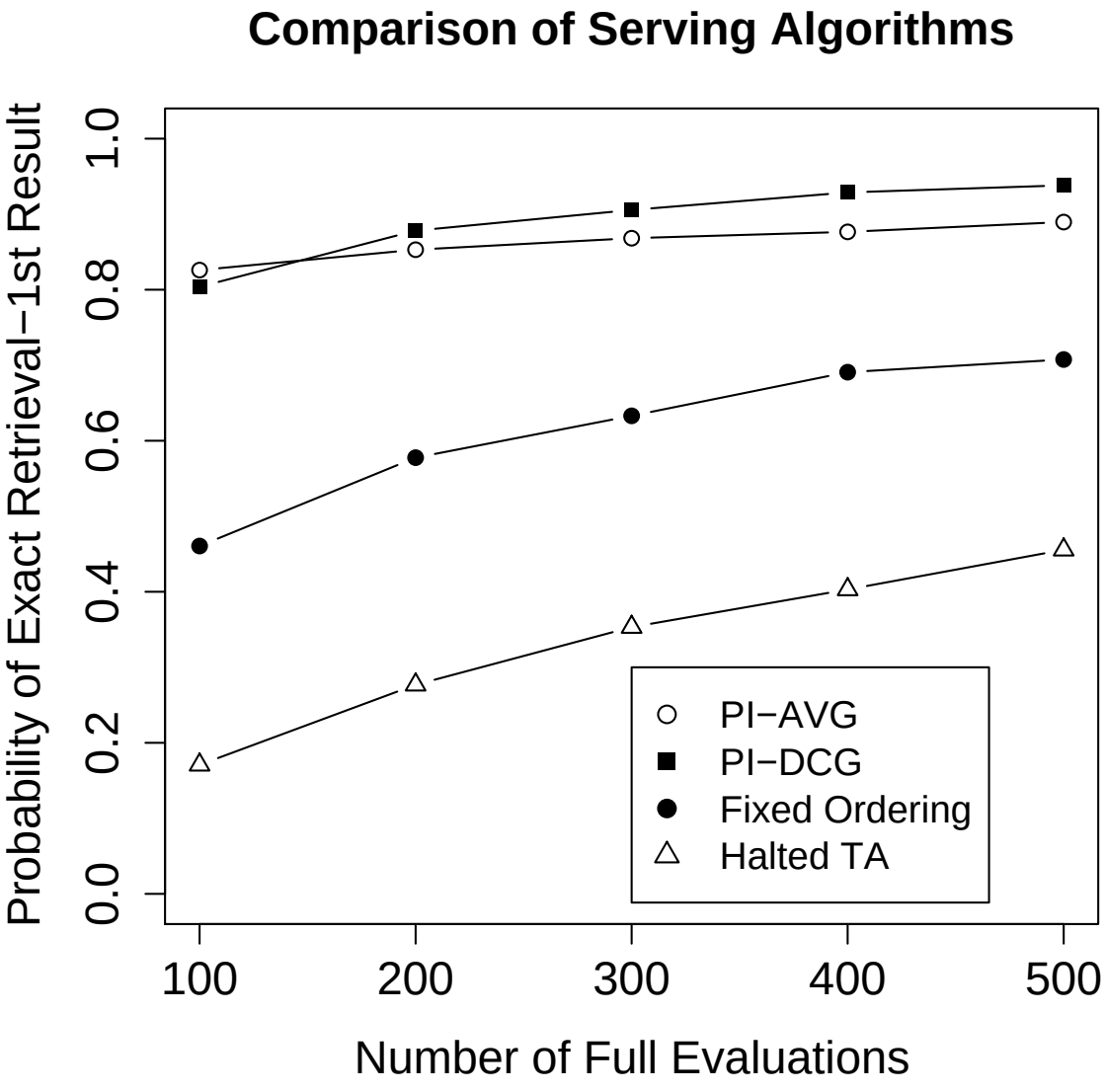
Method 5: Predictive Indexing

For every word/key i construct a sorted list where the list is sorted according to $E[s(q, r)|i \in q]$ or $P(r \texttt{ best}|i \in q]$.

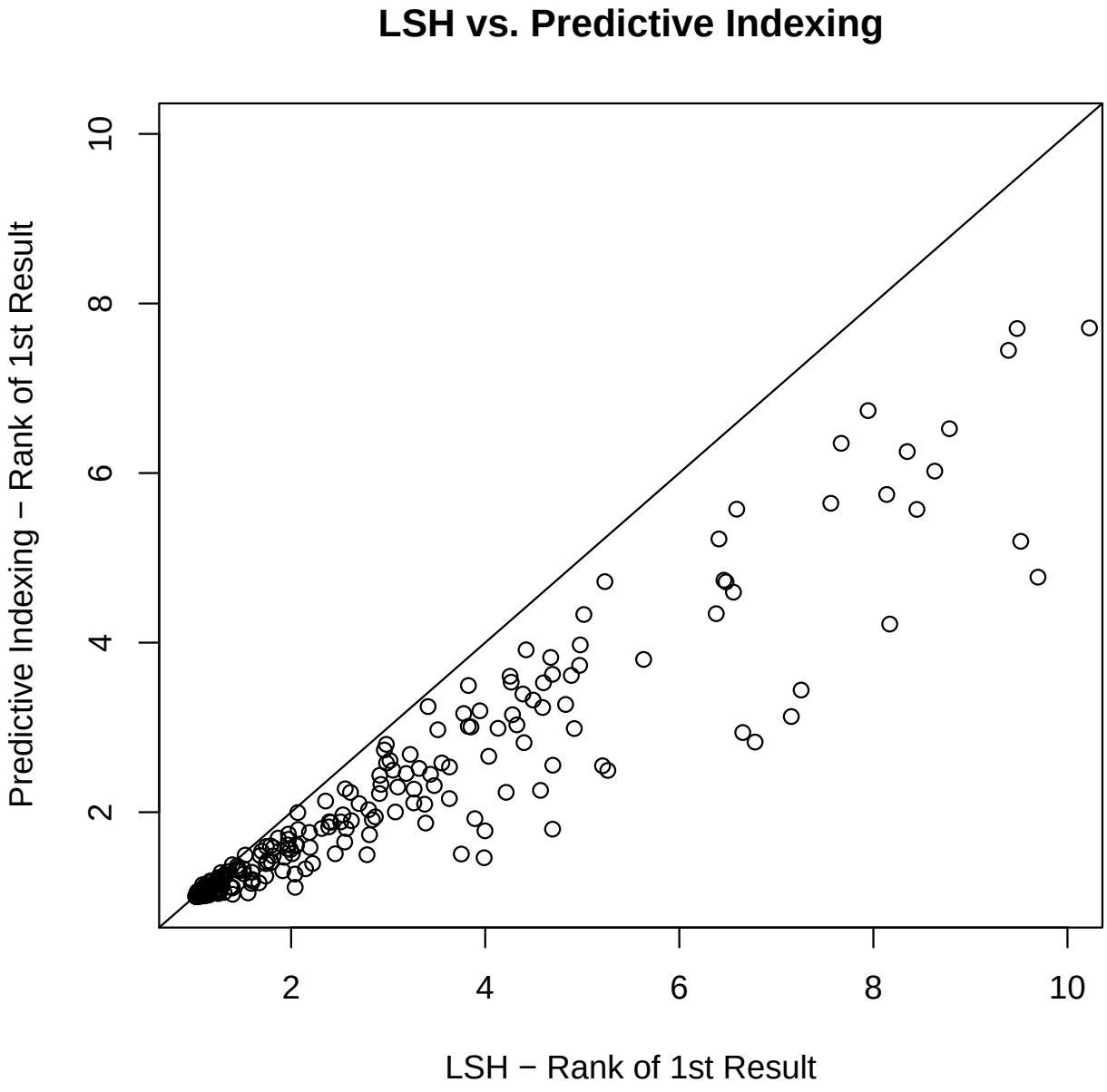
To query, do a breadth first traversal over lists associated with each $i \in q$ doing a full evaluation. When time runs out, return the best result seen.

The ML perspective: scoring directly drives datastructure (good!). Still imperfect—you would prefer the learning algorithm directly learns how to return results efficiently.

Predictive Indexing for an Ad problem



Halted TA = ordering by per-feature score in a linear predictor.



Averaged over many datasets with same random projections used for each.

State of Indexing

Computational efficiency is key here—this is a primary hardware cost.

1. New algorithms can make a big $ difference.
2. Efficient implementations win! FPGAs are tempting.
3. The transition from indexing to scoring is often messy.
4. Data-dependent datastructures are a key improvement.
5. ML often operates as an indexing enhancer.

State of Indexing

Computational efficiency is key here—this is a primary hardware cost.

1. New algorithms can make a big $ difference.
2. Efficient implementations win! FPGAs are tempting.
3. The transition from indexing to scoring is often messy.
4. Data-dependent datastructures are a key improvement.
5. ML often operates as an indexing enhancer.

What is an algorithmically clean and coherent way to learn to index and score simultaneously?

Bibliography

Inverted See Cong Yu's slides for example. CSCI-GA.2580-001 lecture 3.

WAND A Broder, D Carmel, M Herscovici, A Soffer, J Zien. Efficient query evaluation using a two-level retrieval process. CIKM 2003

LSH I A Gionis, P Indyk, R Motwani Similarity search in high dimensions via hashing, VLDB 1999.

LSH II A Andoni, P Indyk. Near-optimal hashing algorithms for near neighbor problem in high dimensions, FOCS 2006.

Predictive S Goel, J Langford and A Strehl, Predictive Indexing for Fast Search, NIPS 2008.