Introduction to

Information Retrieval

Learning to Rank

Machine learning for IR ranking?

- We've looked at methods for ranking documents in IR
  - Cosine similarity, inverse document frequency, BM25, proximity, pivoted document length normalization, (will look at) Pagerank, …
- We've looked at methods for classifying documents using supervised machine learning classifiers
  - Rocchio, kNN, decision trees, etc.
- Surely we can also use *machine learning* to rank the documents displayed in search results?
  - Sounds like a good idea
  - Known as "machine-learned relevance" or "learning to rank"

Senior Machine Learning Scientist

Overstock.com · Midvale, UT, US

Posted 1 day ago · 63 views

Save Apply

Job description

Senior Machine Learning Scientist

The Machine Learning Scientist focuses on core machine learning techniques that include search ranking, recommender systems, natural language processing, computer vision, deep learning, fraud and abuse detection, advertising technologies, personalization and predictive modeling. Our Machine Learning scientists have the opportunity to build cutting-edge e-commerce technologies in all these areas and apply their ideas in different products across our platform. We are looking for individuals who are passionate about machine learning and have a track record as production quality engineers. The Senior Machine Learning Scientist is self-sufficient and can hit the ground running.

Job Responsibilities

- Design and implement core machine learning algorithms used by different product teams, included but not limited to: search ranking, recommender systems, natural language processing, computer vision, deep learning, fraud and abuse detection, advertising technologies, personalization, marketing, CRM and supply chain

Machine learning for IR ranking

- This “good idea” has been actively researched – and actively deployed by major web search engines – in the last 10 years
- Why didn’t it happen earlier?
  - Modern supervised ML has been around for about 25 years…
  - Naïve Bayes has been around for about 60 years…

Machine learning for IR ranking

- There's some truth to the fact that the IR community wasn't very connected to the ML community
- But there were a whole bunch of precursors:
  - Wong, S.K. et al. 1988. Linear structure in information retrieval. *SIGIR 1988.*
  - Fuhr, N. 1992. Probabilistic methods in information retrieval. *Computer Journal.*
  - Gey, F. C. 1994. Inferring probability of relevance using the method of logistic regression. *SIGIR 1994.*
  - Herbrich, R. et al. 2000. Large Margin Rank Boundaries for Ordinal Regression. *Advances in Large Margin Classifiers.*

Why weren't early attempts very successful/influential?

- Sometimes an idea just takes time to be appreciated…
- **Limited training data**
  - Especially for real world use (as opposed to writing academic papers), it was very hard to gather test collection queries and relevance judgments that are representative of real user needs and judgments on documents returned
    - This has changed, both in academia and industry
- Poor machine learning techniques
- Insufficient customization to IR problem
- Not enough features for ML to show value

Why wasn't ML much needed?

- Traditional ranking functions in IR used a very small number of features, e.g.,
  - Term frequency
  - Inverse document frequency
  - Document length
- It was ~~easy~~ possible to tune weighting coefficients by hand
  - And people did

Why is ML needed now?

- Modern (web) systems use a great number of features:
  - Arbitrary useful features – not a single unified model
  - Log frequency of query word in anchor text?
  - Query word in color on page?
  - # of images on page?
  - # of (out) links on page?
  - PageRank of page?
  - URL length?
  - URL contains "~"?
  - Page edit recency?
  - Page loading speed
- The *New York Times* in 2008-06-03 quoted Amit Singhal as saying Google was using over 200 such features ("signals") – so it's sure to be over 500 today. ☺

Simple example: Using classification for ad hoc IR

- Collect a training corpus of (*q, d, r*) triples
  - Relevance *r* is here binary (but may be multiclass, with 3–7 values)
  - Query-Document pair is represented by a feature vector
    - **x** = (α, ω) α is cosine similarity, ω is minimum query window size
      - ω is the the shortest text span that includes all query words
      - Query term proximity is an **important** new weighting factor
  - Train a machine learning model to predict the class *r* of a document-query pair

example	docID	query	cosine score	ω	judgment
Φ_1	37	linux operating system	0.032	3	*relevant*
Φ_2	37	penguin logo	0.02	4	*nonrelevant*
Φ_3	238	operating system	0.043	2	*relevant*
Φ_4	238	runtime environment	0.004	2	*nonrelevant*
Φ_5	1741	kernel layer	0.022	3	*relevant*
Φ_6	2094	device driver	0.03	2	*relevant*
Φ_7	3191	device driver	0.027	5	*nonrelevant*

Simple example: Using classification for ad hoc IR

- A linear score function is then

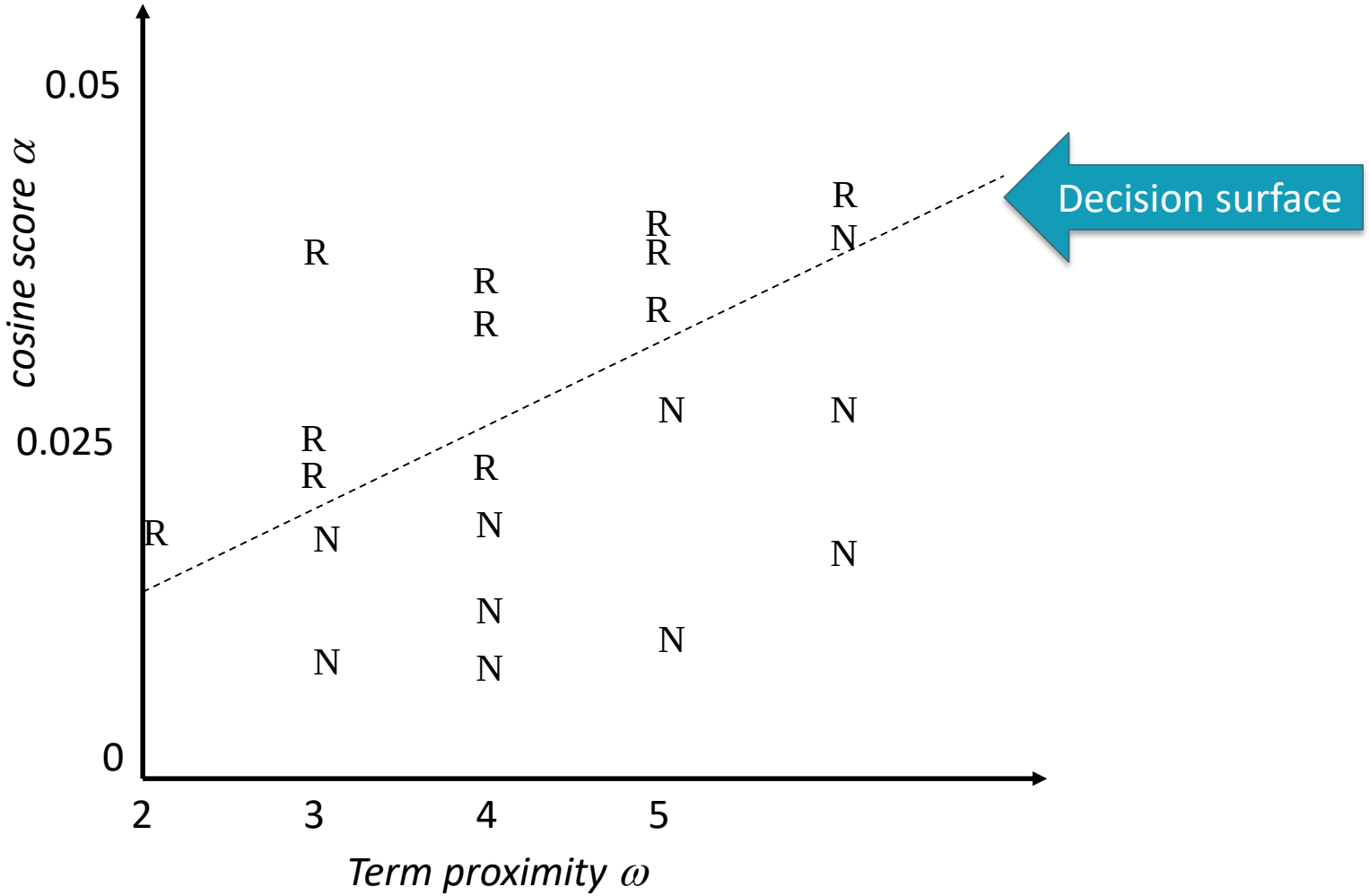
$$Score(d, q) = Score(\alpha, \omega) = a\alpha + b\omega + c$$

- And the linear classifier is

Decide relevant if $Score(d, q) > \theta$

- … just like when we were doing text classification

Simple example: Using classification for ad hoc IR



More complex example of using classification for search ranking [Nallapati 2004]

- We can generalize this to classifier functions over more features
- We can use other methods for learning the linear classifier weights

An SVM classifier for information retrieval

[Nallapati 2004]

- Let relevance score $g(r|d,q) = \mathbf{w} \bullet f(d,q) + b$
- Uses SVM: want $g(r|d,q) \leq -1$ for nonrelevant documents and $g(r|d,q) \geq 1$ for relevant documents
- SVM testing: decide relevant iff $g(r|d,q) \geq 0$

- Features are *not* word presence features (how would you deal with query words not in your training data?) but scores like the summed (log) tf of all query terms
- Unbalanced data (which can result in trivial always-say-nonrelevant classifiers) is dealt with by undersampling nonrelevant documents during training (just take some at random)

An SVM classifier for information retrieval

[Nallapati 2004]

- Experiments:
  - 4 TREC data sets
  - Comparisons with Lemur, a state-of-the-art open source IR engine (Language Model (LM)-based – see *IIR* ch. 12)
  - Linear kernel normally best or almost as good as quadratic kernel, and so used in reported results
  - 6 features, all variants of tf, idf, and tf.idf scores

An SVM classifier for information retrieval [Nallapati 2004]

Train \ Test		Disk 3	Disk 4-5	WT10G (web)
TREC Disk 3	Lemur	**0.1785**	**0.2503**	0.2666
	SVM	0.1728	0.2432	**0.2750**
Disk 4-5	Lemur	**0.1773**	**0.2516**	0.2656
	SVM	0.1646	0.2355	**0.2675**

- At best the results are about equal to Lemur
  - Actually a little bit below
- Paper's advertisement: Easy to add more features
  - This is illustrated on a homepage finding task on WT10G:
    - Baseline Lemur 52% success@10, baseline SVM 58%
    - SVM with URL-depth, and in-link features: 78% success@10

"Learning to rank"

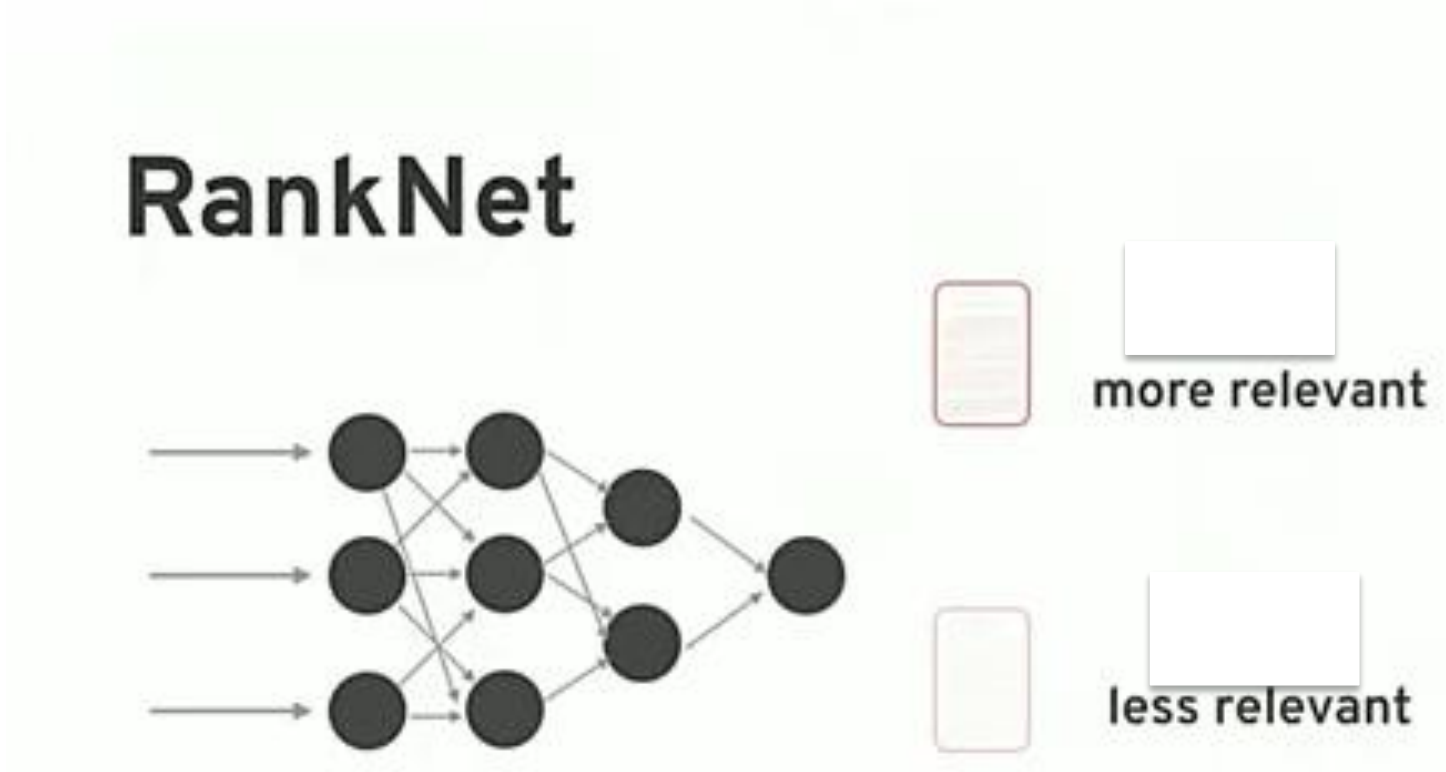
- Classification probably isn't the right way to think about approaching ad hoc IR:
  - Classification problems: Map to an unordered set of classes
  - Regression problems: Map to a real value [See PA3]
  - Ordinal regression (or "ranking") problems: Map to an *ordered* set of classes
    - A fairly obscure sub-branch of statistics, but what we want here
- This formulation gives extra power:
  - Relations between relevance levels are modeled
  - **Documents are good versus other documents for a query given collection**; not an absolute scale of goodness

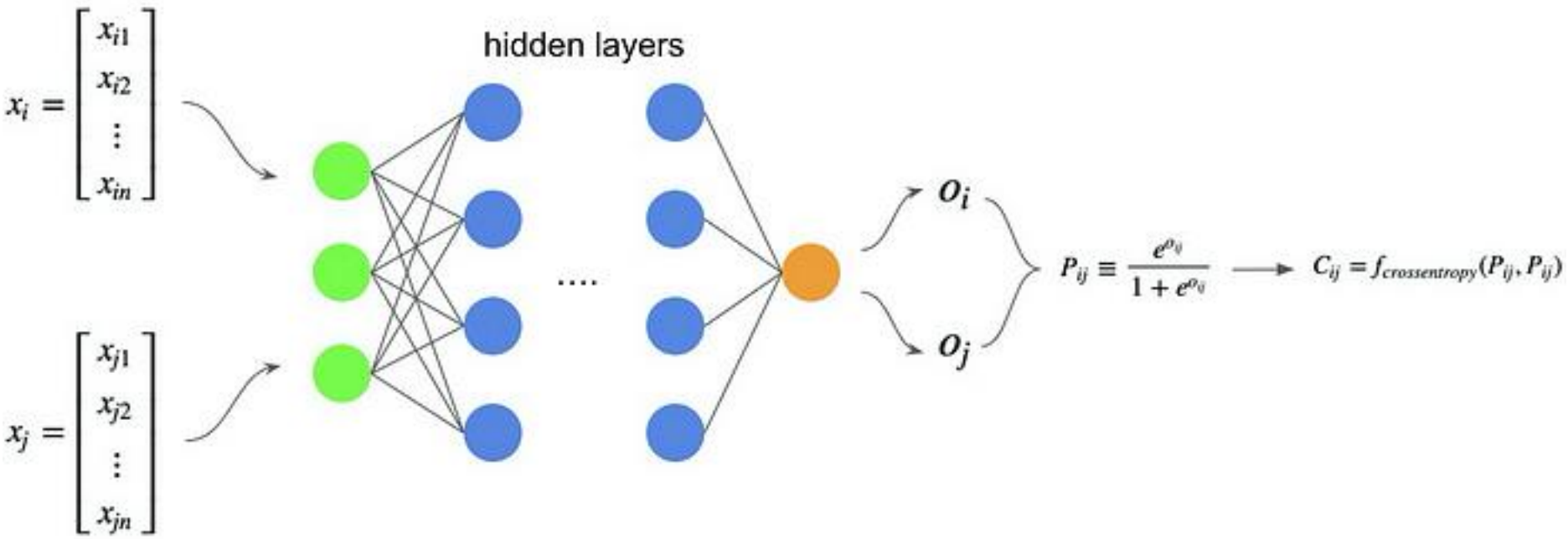
"Learning to rank"

- Assume a number of categories $\boldsymbol{C}$ of relevance exist
  - These are totally ordered: $c_1 < c_2 < \ldots < c_J$
  - This is the ordinal regression setup
- Assume training data is available consisting of document-query pairs (d, q) represented as feature vectors x_i with relevance ranking c_i

Algorithms used for ranking in search

- Support Vector Machines (Vapnik, 1995)
  - Adapted to ranking: Ranking SVM (Joachims 2002)
- Neural Nets: RankNet (Burges et al., 2006)
- Tree Ensembles
  - Random Forests (Breiman and Schapire, 2001)
  - Boosted Decision Trees
    - Multiple Additive Regression Trees (Friedman, 1999)
    - Gradient-boosted decision trees: LambdaMART (Burges, 2010)
    - Used by all search engines? AltaVista, Yahoo!, Bing, Yandex, …
- All top teams in the 2010 Yahoo! Learning to Rank Challenge used combinations with Tree Ensembles!


RankNet
more relevant
less relevant


$x_i = \begin{bmatrix} x_{i1} \\ x_{i2} \\ \vdots \\ x_{in} \end{bmatrix}$
hidden layers
$x_j = \begin{bmatrix} x_{j1} \\ x_{j2} \\ \vdots \\ x_{jn} \end{bmatrix}$
o_i
o_j
$P_{ij} \equiv \frac{e^{o_{ij}}}{1 + e^{o_{ij}}}$
$C_{ij} = f_{crossentropy}(P_{ij}, P_{ij})$

RankNet (a neural net ranker)

- Have differentiable function with model parameters **w**:
  - $\boldsymbol{x}_i \rightarrow f(\boldsymbol{x}; \boldsymbol{w}) = s_i$
- For query q, learn probability of different ranking class for documents $d_i \succ d_j$ via:
  - $P_{ij} = P(d_i \succ d_j) = \frac{1}{1+e^{-\sigma(s_i - s_j)}}$
- Cost function calculates cross entropy loss:
  - $C = -\bar{P}_{ij} \log P_{ij} - (1 - \bar{P}_{ij}) \log(1 - P_{ij})$
- Where P_{ij} is the model probability; $\bar{P}_{ij}$ the actual probability (0 or 1 for categorical judgments)

RankNet (Burges 2010)

- Combining these equations gives
  - $C = \frac{1}{2}(1 - S_{ij})\sigma(s_i - s_j) + \log\left(1 + e^{-\sigma(s_i - s_j)}\right)$
- where, for a given query, $S_{ij} \in \{0, +1, -1\}$
  1 if d_i is more relevant than d_j; –1 if the reverse, and
  0 if the they have the same label
  - $\frac{\partial C}{\partial s_i} = \sigma\left(\frac{1}{2}(1 - S_{ij}) - \frac{1}{1+e^{\sigma(s_i - s_j)}}\right) = -\frac{\partial C}{\partial s_j}$

$$\frac{\partial C}{\partial w_k} = \frac{\partial C}{\partial s_i}\frac{\partial s_i}{\partial w_k} + \frac{\partial C}{\partial s_j}\frac{\partial s_j}{\partial w_k} = \sigma\left(\frac{1}{2}(1 - S_{ij}) - \frac{1}{1 + e^{\sigma(s_i - s_j)}}\right)\left(\frac{\partial s_i}{\partial w_k} - \frac{\partial s_j}{\partial w_k}\right)$$
$$= \lambda_{ij}\left(\frac{\partial s_i}{\partial w_k} - \frac{\partial s_j}{\partial w_k}\right)$$

From RankNet to LambdaRank

- Rather than working with pairwise ranking errors, scale by effect a change has on NDCG
- Idea: Multiply λ's by $|\Delta Z|$, the difference of an IR measure when d_i and d_j are swapped
- E.g. $|\Delta \text{NDCG}|$ is the change in NDCG when swapping d_i and d_j giving:
  - $\lambda_{ij} = \frac{\partial C(s_i - s_j)}{\partial s_i} = \frac{-\sigma}{1+e^{\sigma(s_i - s_j)}} |\Delta \text{NDCG}|$
- Burges et al. "prove" (partly theory, partly empirical) that this change is sufficient for model to optimize NDCG

LambdaRank

- Key observations of LambdaRank:
  - To train a model, we do not need the costs themselves, only the gradients (of the costs. wrt model scores).
  - The gradient should be bigger for pairs of documents that produces a bigger impact in NDCG by swapping positions.
- LambdaRank [Burges et al., 2006] Multiply actual gradients with the change in NDCG by swapping the rank positions of the two documents:
- LambdaRank = RankNet $\cdot |\Delta \text{NDCG}|$
-

$$\lambda_{\text{LambdaRank}} = \lambda_{\text{RankNet}} \cdot |\Delta \text{NDCG}|$$

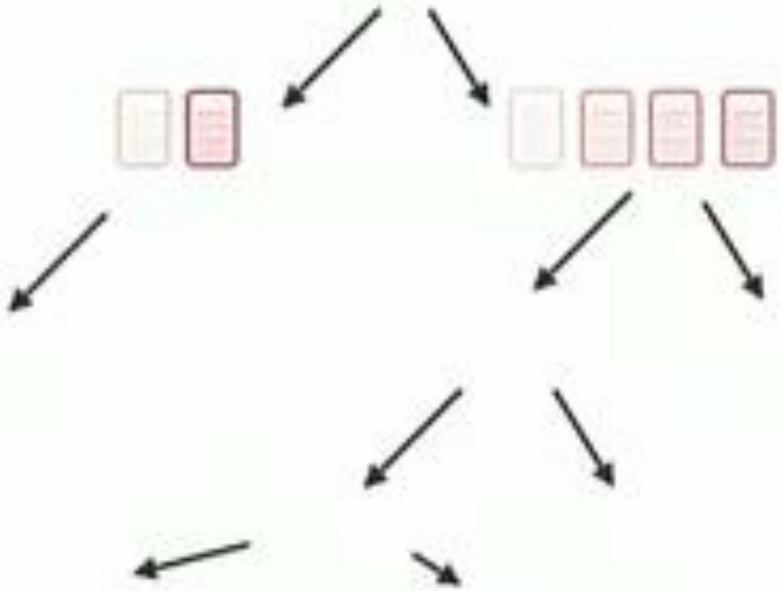
From LambdaRank to LambdaMART

- LambdaRank models gradients
- MART can be trained with gradients ("gradient boosting")
- Combine both to get LambdaMART
  - MART with specified gradients and optimization step

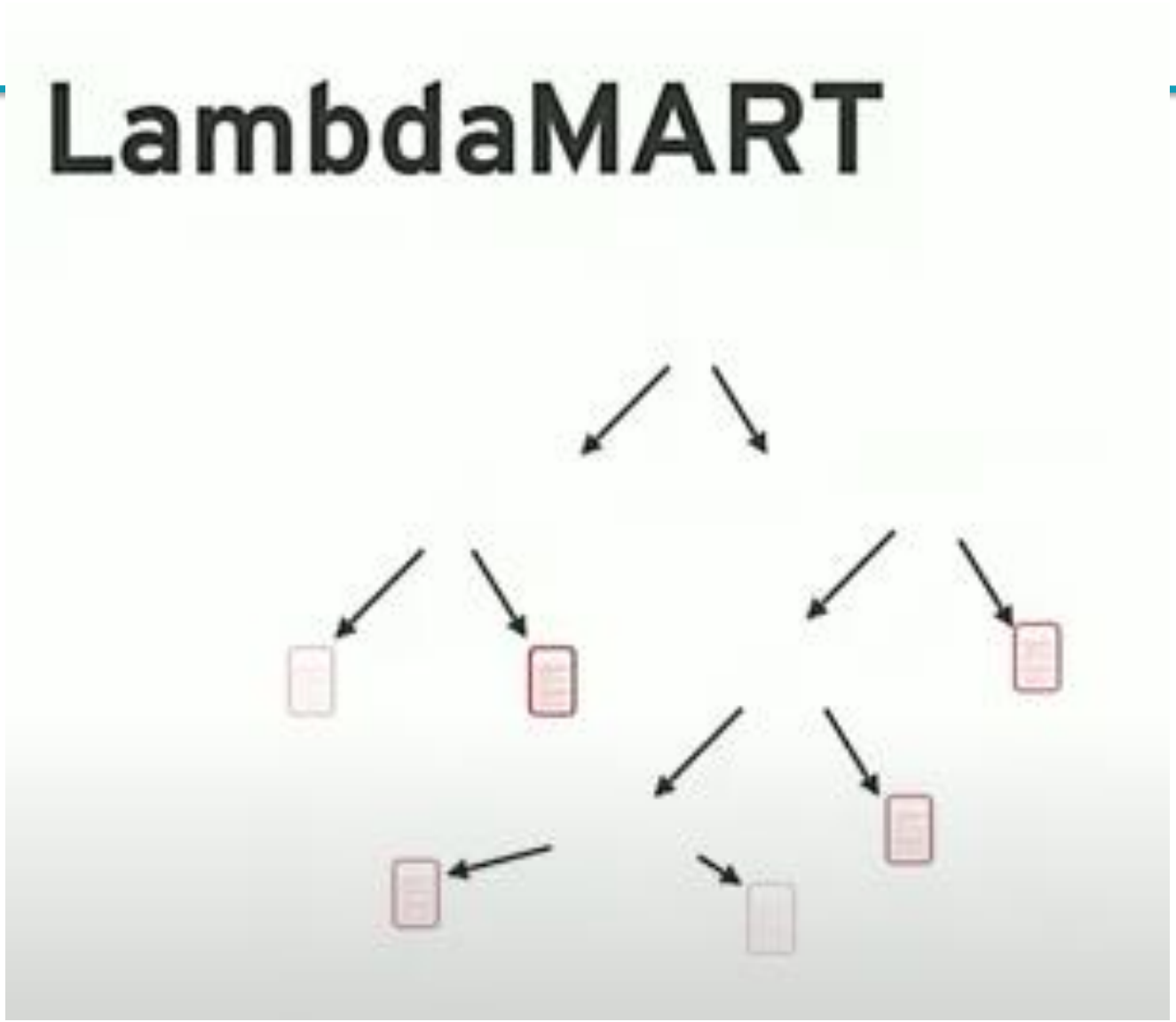
LambdaMART

LambdaMART is a combination of LambdaRank and MART (Multiple Additive Regression Trees). MART uses gradient boosted decision trees for prediction tasks. However, LambdaMART improves this by using gradient boosted decision trees with a cost function derived from LambdaRank to order any ranking situation. LambdaMART has shown better results than LambdaRank and the original RankNet on experimental datasets.

LambdaMART



LambdaMART



Regression trees

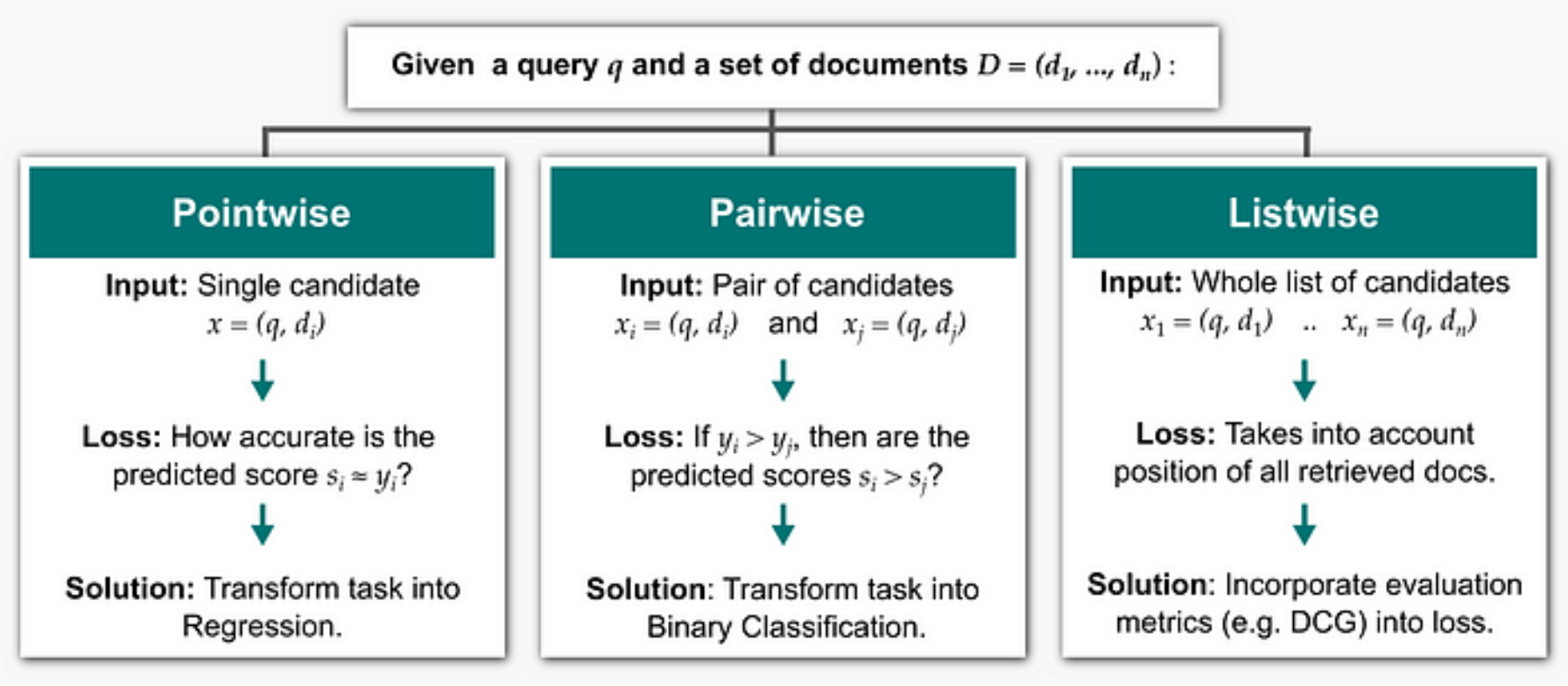
- Decision trees can predict a real value
  - They're then often called "regression trees"
- The value of a leaf node is the mean of all instances at the leaf $\gamma_k = f(x_i) = \overline{x_i}$
- Splitting criterion: Standard Deviation Reduction
  - Choose split value to minimize the variance (standard deviation SD) of the values in each subset S_i of S induced by split A (normally just a binary split for easy search):
    - $SDR(A,S) = SD(S) - \sum_i \frac{|S_i|}{|S|} SD(S_i)$
    - $SD = \sum_i (y_i - f(x_i))^2$
- Termination: cutoff on SD or #examples or tree depth

General Types of LTR Algorithms

1. Pointwise
2. Pairwise
3. Listwise

They are distinguished by how we formulate the **loss function** in the underlying machine learning task.

Learning to Rank



Pointwise Method

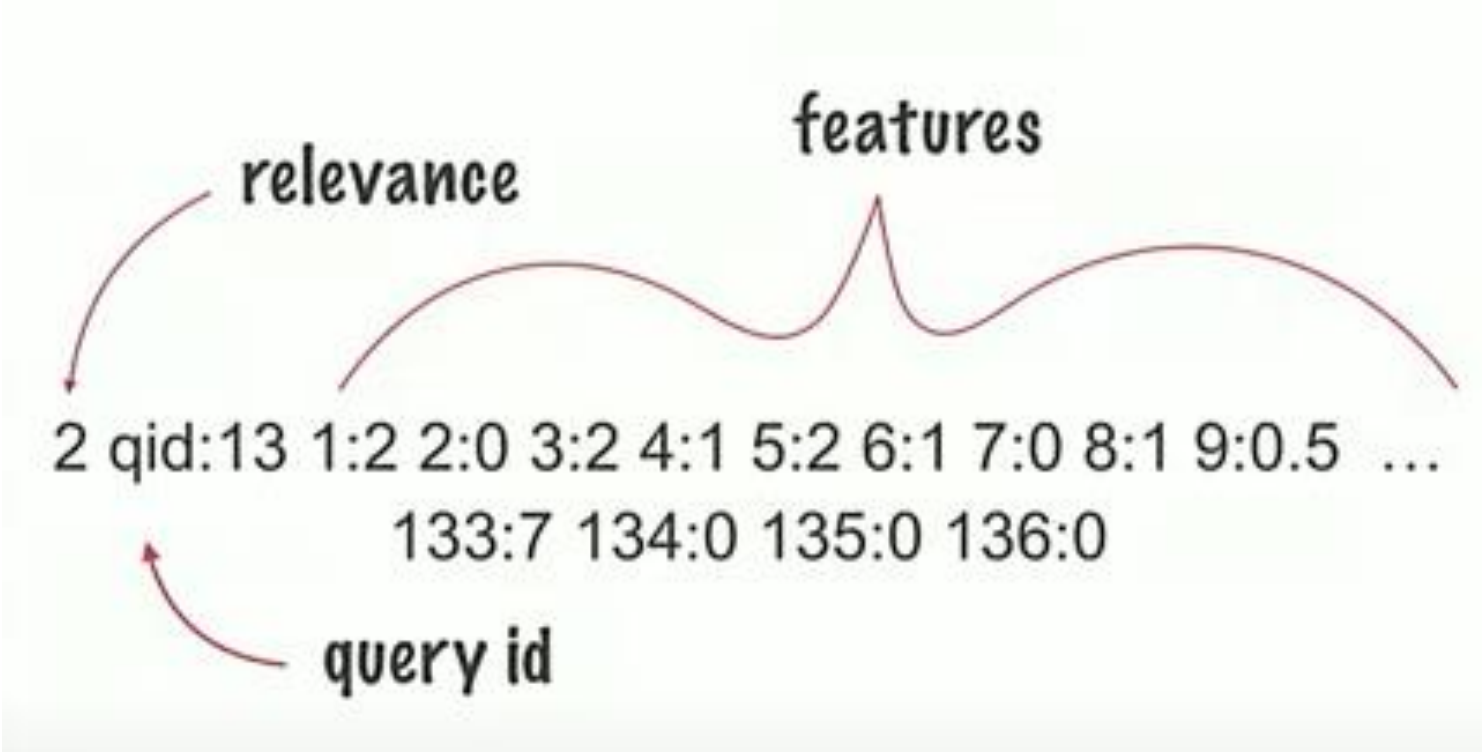
- Pros:
  - Simplicity. Existing ML models are ready to apply.
- Cons:
  - The result is usually sub-optimal due to not utilizing the full information in the entire list of matching documents for each query.
  - Explicit pointwise labels are required to constitute the training dataset.

Pairwise Method (RankNet, LambdaRank, LambdaMART)

- Pros:
  - The model is learning how to rank directly, even though only in a pairwise manner, but in theory it can approximate the performance of a general ranking task given N document in a matched list.
  - We don't need explicit pointwise labels. Only pairwise preferences are required. This is an advantage because sometimes we are only able to infer the pairwise preference from collected user behavior.
- Cons:
  - Scoring function itself is still pointwise, meaning that relative information in the feature space among different documents given the same query is still not fully exploited.

MSLR-WEB10K

- Open Source
- 10,000 queries
- relevance values from 0 to 4
- schema: relevance, query id, feature vector
- training, testing and validation data sets


features
relevance
2 qid:13 1:2 2:0 3:2 4:1 5:2 6:1 7:0 8:1 9:0.5 …
133:7 134:0 135:0 136:0
query id

Summary

- The idea of learning ranking functions has been around for about 20 years
- But only more recently have ML knowledge, availability of training datasets, a rich space of features, and massive computation come together to make this a hot research area
- It's too early to give a definitive statement on what methods are best in this area
- But machine-learned ranking over many features now easily beats traditional hand-designed ranking functions in comparative evaluations [in part by using the hand-designed functions as features!]
- There is every reason to think that the importance of machine learning in IR will grow in the future.

Resources

- *IIR* secs 6.1.2–3 and 15.4
- Nallapati, R. Discriminative models for information retrieval. *SIGIR 2004.*
- LETOR benchmark datasets
  - Website with data, links to papers, benchmarks, etc.
  - http://research.microsoft.com/users/LETOR/
  - Everything you need to start research in this area! But smallish.
- C. J. C. Burges. From RankNet to LambdaRank to LambdaMART: An Overview. Microsoft TR 2010.
- O. Chapelle and Y. Chang. Yahoo! Learning to Rank Challenge Overview. *JMLR Proceedings* 2011.