Introduction to
Information Retrieval

CS276: Information Retrieval and Web Search
Christopher Manning and Pandu Nayak

Lecture 14: Learning to Rank (with GBDTs)

Borrows slides/pictures from Schigehiko Schamoni

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
  - You students do it in PA3

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

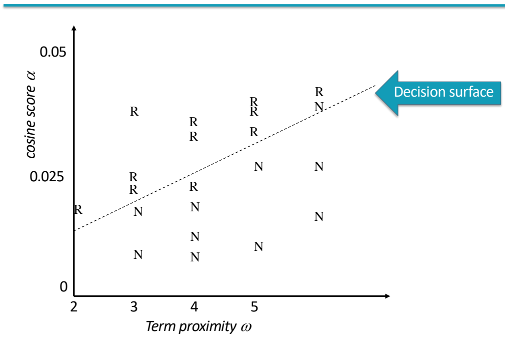
$$Score(d, q) = Score(\alpha, \omega) = a\alpha + b\omega + c$$

- 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 ***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!

Yahoo! Learning to Rank Challenge

(Chapelle and Chang, 2011)

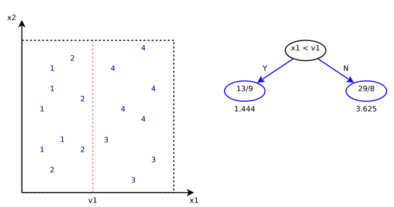
- Yahoo! Webscope dataset : 36,251 queries, 883k documents, 700 features, 5 ranking levels
  - Ratings: Perfect (navigational), Excellent, Good, Fair, Bad
  - Real web data from U.S. and "an Asian country"
  - set-1: 473,134 feature vectors; 519 features; 19,944 queries
  - set-2: 34,815 feature vectors; 596 features; 1,266 queries
- Winner (Burges et al.) was linear combo of 12 models:
  - 8 Tree Ensembles (LambdaMART)
  - 2 LambdaRank Neural Nets
  - 2 Logistic regression models

Regression trees

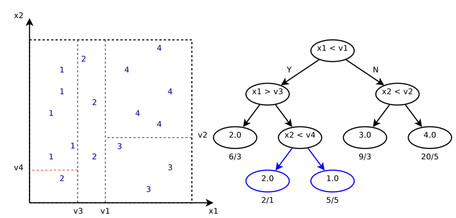
- Decision trees can predict a real value
  - They're then often called "regression trees"
- The value of a leaf node is the mean of all instances at the leaf $\gamma_k = f(x_i) = x_i$
- Splitting criterion: Standard Deviation Reduction
  - Choose split value to minimize the variance (standard deviation 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

Training a regression tree

- The algorithm searches for split variables and split points, x_1 and v_1 so as to minimize the predicted error, i.e., $\sum_i (y_i - f(x_i))^2$.



- You can grow tree till 0 error (if no identical points with different scores)



- 3d e.g.: http://arogozhnikov.github.io/2016/06/24/gradient_boosting_explained.html

The concept of boosting

- Motivating question:
  - Can we use individually weak machine learning classifiers to build a high-accuracy classification system?
- Classic approach (AdaBoost)
  - Learn a small decision tree (often a 1-split decision stump)
  - It will get the biggest split in the data right
  - Repeat:
    - Upweight examples it gets wrong;
    - Downweight examples it gets right
    - Learn another small decision tree on that reweighted data
- Classify with weighted vote of all trees
  - Weight trees by individual accuracy

Towards gradient boosting: Function estimation

- Want: a function $F^*(x)$ that maps ***x*** to *y*, s.t. the expected value of some loss function L(y, F(**x**)) is minimized:
  - $F^*(\boldsymbol{x}) = \arg\min_{F(\boldsymbol{x})} \mathbb{E}_{y,\boldsymbol{x}} L(y, F(\boldsymbol{x}))$
- Boosting approximates $F^*(x)$ by an additive expansion
  - $F(\boldsymbol{x}) = \sum_{m=1}^{M} \beta_m h(\boldsymbol{x}; \boldsymbol{a}_m)$
- where $h(\mathbf{x}; \boldsymbol{a})$ are simple functions of ***x*** with parameters $\boldsymbol{a} = \{a_1, a_2, \ldots, a_n\}$ defining the function h, and the β are weighting coefficients

Finding parameters

- Function parameters are iteratively fit to the training data:
  - Set $F_0(\mathbf{x})$ = initial guess (or zero)
  - For each m = 1, 2, …, M
    - $\boldsymbol{a}_m = \arg\min_{\boldsymbol{a}} \sum_i L(y_i, F_{m-1}(\boldsymbol{x}_i) + \beta h(\boldsymbol{x}_i, \boldsymbol{a}))$
    - $F_m(\boldsymbol{x}) = F_{m-1}(\boldsymbol{x}_i) + \beta h(\boldsymbol{x}_i, \boldsymbol{a})$
- You successively estimate and add a new tree to the sum
- You never go back to revisit past decisions

Finding parameters

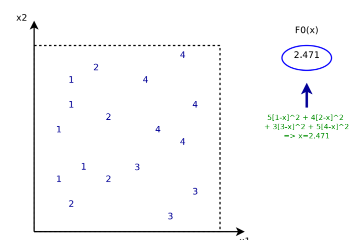
- Gradient boosting approximately achieves this for *any* differentiable loss function
  - Fit the function h(x; a) by least squares
    - $\boldsymbol{a}_m = \arg\min_a \sum_i [\tilde{y}_{im} - h(\boldsymbol{x}_i, \boldsymbol{a})]^2$
  - to the "pseudo-residuals" (deviation from desired scores)
    - $\tilde{y}_{im} = -\left[\frac{\partial L(y_i, F(\boldsymbol{x}_i))}{\partial F(\boldsymbol{x}_i)}\right]_{F(\boldsymbol{x}) = F_{m-1}(\boldsymbol{x})}$
- Whatever the loss function, gradient boosting simplifies the problem to least squares estimation!!!
  - We can take a gradient (Newton) step to improve model

Gradient tree boosting

- Gradient tree boosting applies this approach on functions $h(\mathbf{x}; \boldsymbol{a})$ which are small regression trees
  - The trees used normally have 1–8 splits only
  - Sometimes stumps do best!
  - The allowed depth of the tree controls the feature interaction order of model (do you allow feature pair conjunctions, feature triple conjunctions, etc.?)

Learning a gradient-boosted regression tree

- First, learn the simplest predictor that predicts a constant value that minimizes the error on the training data



Learning a gradient-boosted regression tree

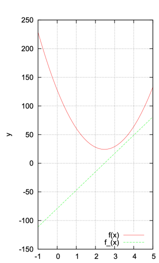
- We want to find value γ_{km} for root node of tree

Quadratic loss for the leaf (red):

$$f(x) = 5 \cdot (1-x)^2 + 4 \cdot (2-x)^2 + 3 \cdot (3-x)^2 + 5 \cdot (4-x)^2$$

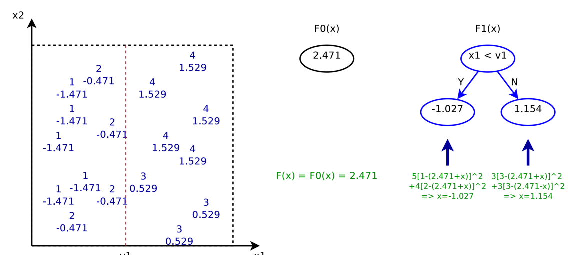
$f(x)$ is quadratic, *convex*
$\Rightarrow$ Optimum at $f'(x) = 0$ (green)

$$\frac{\partial f(x)}{\partial x} = 5 \cdot (-2+2x) + 4 \cdot (-4+2x) + 3 \cdot (-6+2x)^2 + 5 \cdot (-8+2x)^2 = -84 + 34x = 34(x - 2.471)$$



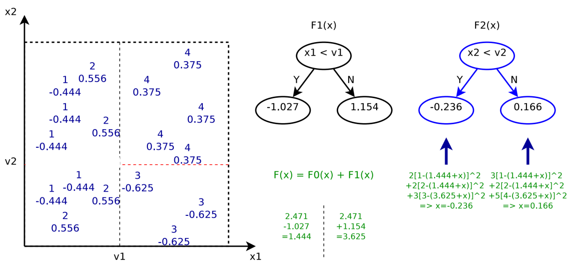
Learning a gradient-boosted regression tree

- We split root node based on least squares criterion and build a tree predicting "pseudo-residuals"



Learning a gradient-boosted regression tree

- Then another tree is added to fit the actual "pseudo-residuals" of the first tree



Multiple Additive Regression Trees (MART) [Friedman 1999]

Algorithm 1 Multiple Additive Regression Trees.

Initialize $F_0(\mathbf{x}) = \arg\min_\gamma \sum_{i=1}^{N} L(y_i, \gamma)$
for $m = 1, \ldots, M$ **do**
  for $i = 1, \ldots, N$ **do**
    $\tilde{y}_{im} = -\left[\frac{\partial L(y_i, F(\mathbf{x}_i))}{\partial F(\mathbf{x}_i)}\right]_{F(\mathbf{x}) = F_{m-1}(\mathbf{x})}$
  end for
  $\{R_{km}\}_{k=1}^{K}$ // Fit a regression tree to targets $\tilde{y}_{im}$
  for $k = 1, \ldots, K_m$ **do**
    $\gamma_{km} = \arg\min_\gamma \sum_{\mathbf{x}_i \in R_{jm}} L(y_i, F_{m-1}(\mathbf{x}_i) + \gamma)$
  end for
  $F_m(\mathbf{x}) = F_{m-1}(\mathbf{x}) + \eta \sum_{k=1}^{K_m} \gamma_{km} 1(\mathbf{x}_i \in R_{km})$
end for
Return $F_M(\mathbf{x})$

Historical path to LambdaMART: via RankNet (a neural net ranker)

- Have differentiable function with model parameters **w**:
  - $\boldsymbol{x_i} \to f(\boldsymbol{x}; \boldsymbol{w}) = s_i$
- 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:
  - $C = -P_{ij} \log P_{ij} - (1 - P_{ij}) \log(1 - P_{ij})$
- Where P_{ij} is the model probability; 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
  - $\frac{\partial C}{\partial s_i} = \sigma\left(\frac{1}{2}(1 - S_{ij}) - \frac{1}{1+e^{\sigma(s_i - s_j)}}\right) = -\frac{\partial C}{\partial s_j}$

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

RankNet lambdas

- The crucial part of the update is

$$\frac{\partial C}{\partial w_k} = \frac{\partial C}{\partial s_i}\frac{\partial s_i}{\partial w_k} + \frac{\partial C}{\partial s_j}\frac{\partial s_j}{\partial w_k} = \lambda_{ij}\left(\frac{\partial s_i}{\partial w_k} - \frac{\partial s_j}{\partial w_k}\right)$$

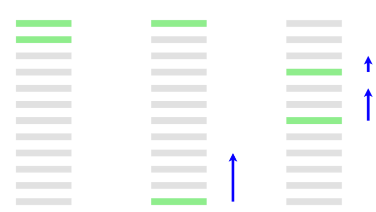
- λ_{ij} describes the desired change of scores for the pair of documents d_i and d_j
- The sum of all λ_{ij}'s and λ_{ji}'s of a query-doc vector x_i w.r.t. all other differently labelled documents for q is

$$\lambda_i = \sum_{j:\{i,j\} \in I} \lambda_{ij} - \sum_{k:\{k,i\} \in I} \lambda_{ki}$$

- λ_i is (sort of) a gradient of the pairwise loss of vector x_i

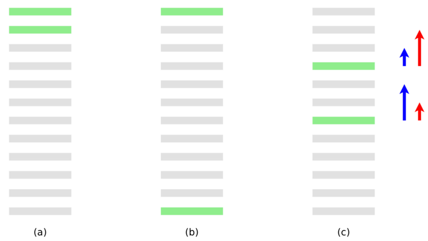
RankNet lambdas (Burges 2010)

- (a) is the perfect ranking, (b) is a ranking with 10 pairwise errors, (c) is a ranking with 8 pairwise errors. Each blue arrow represents the λ_i for each query-document vector x_i



RankNet lambdas (Burges 2010)

- Problem: RankNet is based on pairwise error, while modern IR measures emphasize higher ranking positions. Red arrows show better λ's for modern IR, esp. web search.



From RankNet to LambdaRank

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

set number of trees N, number of training samples m, number of leaves per tree L, learning rate η

for $i = 0$ to m **do**

  $F_0(x_i) = \mathrm{BaseModel}(x_i)$ //If BaseModel is empty, set $F_0(x_i) = 0$

end for

for $k = 1$ to N **do**

  for $i = 0$ to m **do**

    $y_i = \lambda_i$

    $w_i = \frac{\partial y_i}{\partial F_{k-1}(x_i)}$

  end for

  $\{R_{lk}\}_{l=1}^{L}$ // Create L leaf tree on $\{x_i, y_i\}_{i=1}^{m}$ R_{lk} is data items at leaf node l

  $\gamma_{lk} = \frac{\sum_{x_i \in R_{lk}} y_i}{\sum_{x_i \in R_{lk}} w_i}$ // Assign leaf values based on Newton step.

  $F_k(x_i) = F_{k-1}(x_i) + \eta \sum_l \gamma_{lk} I(x_i \in R_{lk})$ // Take step with learning rate η.

end for

Yahoo! Learning to rank challenge

- Goal was to validate learning to rank methods on a large, "real" web search problem
  - Previous work was mainly driven by LETOR datasets
    - Great as first public learning-to-rank data
    - Small: 10s of features, 100s of queries, 10k's of docs
- Only feature vectors released
  - Not URLs, queries, nor feature descriptions
    - Wanting to keep privacy and proprietary info safe
  - But included web graph features, click features, page freshness and page classification features as well as text match features

Burges et al. (2011) entry systems

Model	Description	ERR	NDCG
M1	LambdaMART optimized for ERR	0.461	0.774
M2	LambdaMART optimized for ERR trained on Aug50	0.464	0.786
M3	LambdaMART optimized for ERR trained on Aug70	0.462	0.780
M4	LambdaMART trained for ERR with MART scores as features	0.460	0.775
M5	LambdaMART optimized for NDCG	0.462	0.779
M6	LambdaMART optimized for NDCG trained on Aug50	0.464	0.787
M7	LambdaMART optimized for NDCG trained on Aug70	0.463	0.783
M8	LambdaMART trained for NDCG with MART scores as features	0.461	0.781
M9	LambdaRank optimized for ERR	0.453	0.750
M10	LambdaRank optimized for NDCG	0.453	0.757
M11	MART	0.455	0.772
M12	MART with output scores normalized to unit variance per query	0.455	0.772

ERR = Expected reciprocal rank; see Chapelle and Chang (2011)

They didn't need to combine so many

Ensemble	ERR	NDCG
All: M1–M12	0.4657	0.7878
All minus MARTs: M1–M10	0.4657	0.7875
LambdaMART only: M1–M8	0.4652	0.7874
All minus LambdaRanks: M1–M8, M11, M12	0.4653	0.7879
Augmented only: M2, M3, M6, M7	0.4641	0.7864
Non-augmented: M1, M4, M5, M8–M12	0.4648	0.7850
Just ERR: M1–M3, M4, M9	0.4657	0.7860
Just NDCG: M5–M8, M10	0.4652	0.7869

All good systems performed almost identically, trained on the same features

Table 4: ERR and NDCG accuracies on the final validation and test data.

Rank	Team	Test ERR	Test NDCG	Valid ERR	Valid NDCG
1	Ca3Si2O7	0.468605	0.8041	0.4611	0.7995
2	catonakeyboardinspace	0.467857	0.8060	0.4609	0.8011
3	MLG	0.466954	0.8026	0.4600	0.7960
4	Joker	0.466776	0.8053	0.4607	0.8011
5	AG	0.466157	0.8018	0.4606	0.8010

It's not very clear that you need to use LambdaMART.
Methods like (pairwise) Logistic Rank seem to do just fine.
But use of trees seems to be de rigeur at search engine companies.
So maybe they're a little better on big data?

Raw example of xgboost for ranking with LambdaMART

- https://github.com/dmlc/xgboost/tree/master/demo/rank
- git clone https://github.com/dmlc/xgboost.git
- brew install unrar #somehow get unrar if don't have it
- cd gboost/demo/rank
- ./wgetdata.sh # gets one of the LETOR datasets
- python notebook

Raw example of xgboost for ranking with LambdaMART

```
In [1]: import xgboost as xgb
        from xgboost import DMatrix
        from sklearn.datasets import load_svmlight_file

        # Load the data: It's in svmlight format lines: class featnum:val ...
        x_train, y_train = load_svmlight_file("mq2008.train")
        x_valid, y_valid = load_svmlight_file("mq2008.vali")
        x_test, y_test = load_svmlight_file("mq2008.test")
```


Raw example of xgboost for ranking with LambdaMART

```
In [4]: # MUST use groups of data items from same query to get IR ranking to work!
        group_train = []
        with open("mq2008.train.group", "r") as f:
            data = f.readlines()
            for line in data:
                group_train.append(int(line.split("\n")[0]))

        group_valid = []
        with open("mq2008.vali.group", "r") as f:
            data = f.readlines()
            for line in data:
                group_valid.append(int(line.split("\n")[0]))

        group_test = []
        with open("mq2008.test.group", "r") as f:
            data = f.readlines()
            for line in data:
                group_test.append(int(line.split("\n")[0]))


        train_dmatrix = DMatrix(x_train, y_train)
        valid_dmatrix = DMatrix(x_valid, y_valid)
        test_dmatrix = DMatrix(x_test)

        train_dmatrix.set_group(group_train)
        valid_dmatrix.set_group(group_valid)
        test_dmatrix.set_group(group_test)
```


Raw example of xgboost for ranking with LambdaMART

```
In [38]: # Use rank:pairwise. eta is step size (between 0.01 and 0.1 probably); limit tree depth
         # The parameters I use here are different to the ones provided, and seem better (IMHO)
         # I track validation performance with ndcg@3
         params = {'objective': 'rank:pairwise', 'eta': 0.05, 'gamma': 1.0,
                   'min_child_weight': 0.1, 'max_depth': 2, 'eval_metric': 'ndcg@3'}
         xgb_model = xgb.train(params, train_dmatrix, num_boost_round=200,
                               evals=[(valid_dmatrix, 'validation')])

         [0]     validation-ndcg@3:0.661024
         [1]     validation-ndcg@3:0.669619
         [2]     validation-ndcg@3:0.669619
         [3]     validation-ndcg@3:0.669619
         [4]     validation-ndcg@3:0.669619
         [5]     validation-ndcg@3:0.669619
         [6]     validation-ndcg@3:0.673638
         [7]     validation-ndcg@3:0.673638
         [8]     validation-ndcg@3:0.673638
         [9]     validation-ndcg@3:0.671664
         [10]    validation-ndcg@3:0.671664
         [11]    validation-ndcg@3:0.671664
         [12]    validation-ndcg@3:0.671664
```

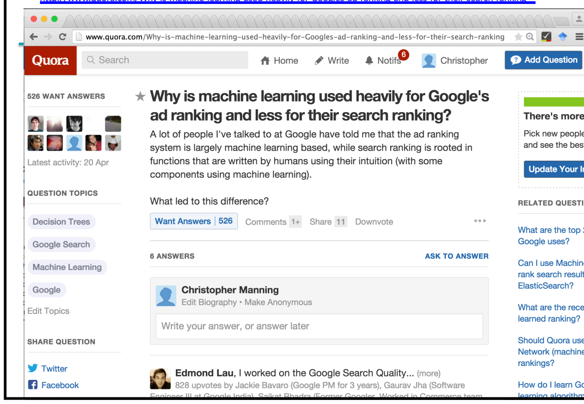
Raw example of xgboost for ranking with LambdaMART

```
[186]   validation-ndcg@3:0.691521
[187]   validation-ndcg@3:0.691521
[188]   validation-ndcg@3:0.691521
[189]   validation-ndcg@3:0.692755
[190]   validation-ndcg@3:0.692553
[191]   validation-ndcg@3:0.692041
[192]   validation-ndcg@3:0.691207
[193]   validation-ndcg@3:0.691207
[194]   validation-ndcg@3:0.689713
[195]   validation-ndcg@3:0.689713
[196]   validation-ndcg@3:0.689713
[197]   validation-ndcg@3:0.68975
[198]   validation-ndcg@3:0.691244
[199]   validation-ndcg@3:0.691244
```

```
In [21]: pred = xgb_model.predict(test_dmatrix)
         pred
         # You would have to do some work with these, using group_test
         # to evaluate for NDCG

Out[21]: array([ 1.3145036 , -0.9180857 ,  1.3107703 , ...,  0.7397013 ,
                 1.2283247 , -0.46511227], dtype=float32)
```


http://www.quora.com/Why-is-machine-learning-used-heavily-for-Googles-ad-ranking-and-less-for-their-search-ranking



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.