MLR: Machine Learning Ranking

IIR secs 6.1.2–3 and 15.4

SAPIENZA
UNIVERSITÀ DI ROMA

Fabrizio Silvestri

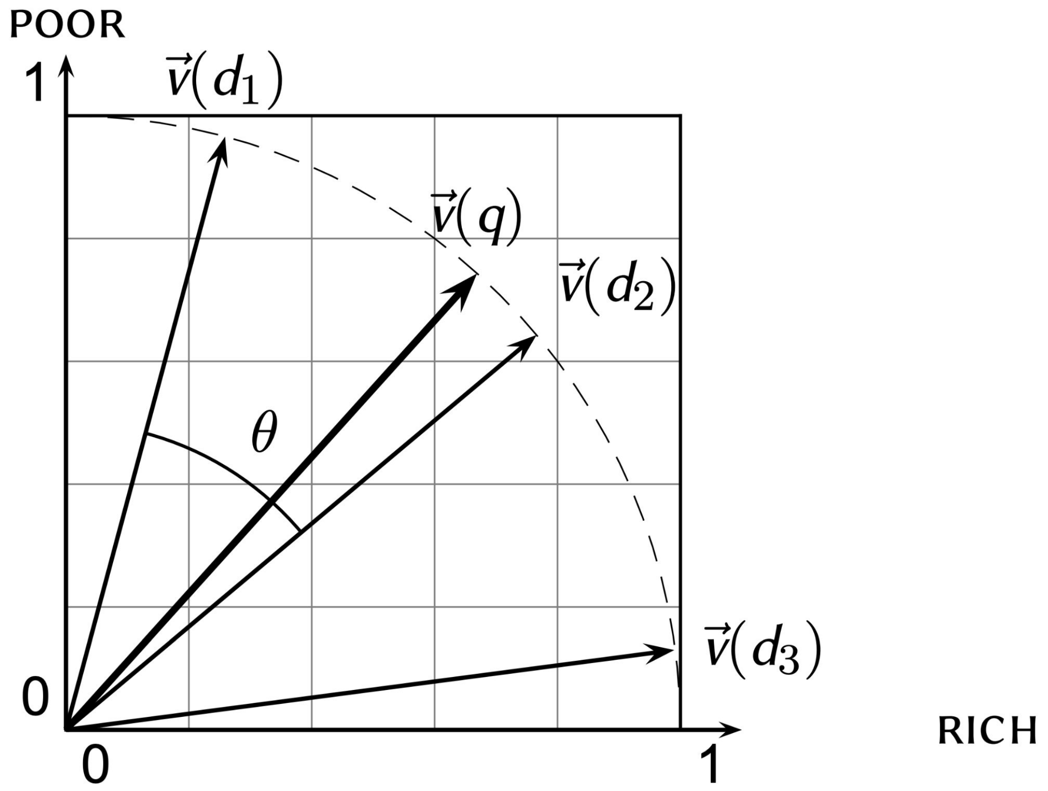
Quick Intro

Cosine Similarity between Query and Document

$$\cos(\vec{q},\vec{d}) = \textsc{sim}(\vec{q},\vec{d}) = \frac{\vec{q}}{|\vec{q}|}\cdot\frac{\vec{d}}{|\vec{d}|} = \sum_{i=1}^{|V|}\frac{q_i}{\sqrt{\sum_{i=1}^{|V|} q_i^2}}\cdot\frac{d_i}{\sqrt{\sum_{i=1}^{|V|} d_i^2}}$$

- q_i is the tf-idf weight of term *i* in the query.
- d_i is the tf-idf weight of term *i* in the document.
- |***q***| and |***d***| are the lengths of vectors ***q*** and ***d****, respectively*.
- ***q***/|***q***| and ***d***/|***d***| are length-1 vectors (= normalized)

Computing Scores

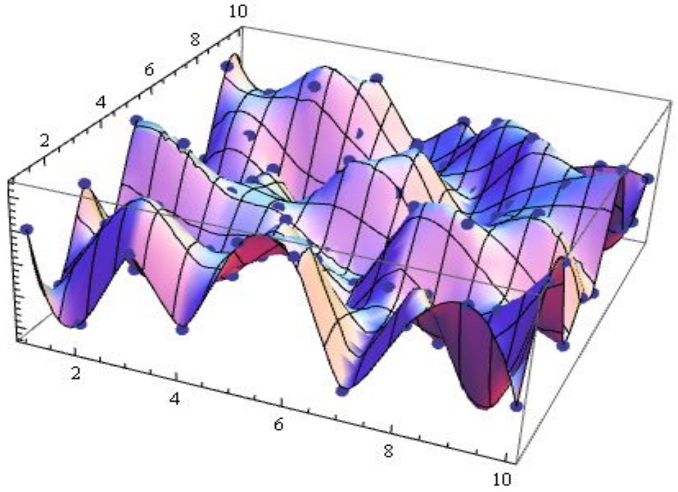


Other factors affecting scores

- Query-Click pairs
- Different sections of a document have different importance
- Date, time of query affects score
- Location of the searcher
- Demographic information
- History of queries
- History of clicks
- History of visited pages
- ...

Why cosine similarity?

- BM25 (a refined form of cosine similarity) arises from well defined principles.
- But… score = f(query, document)
  - What is query? And document?
  - What is the functional form of f?
- Interpolation

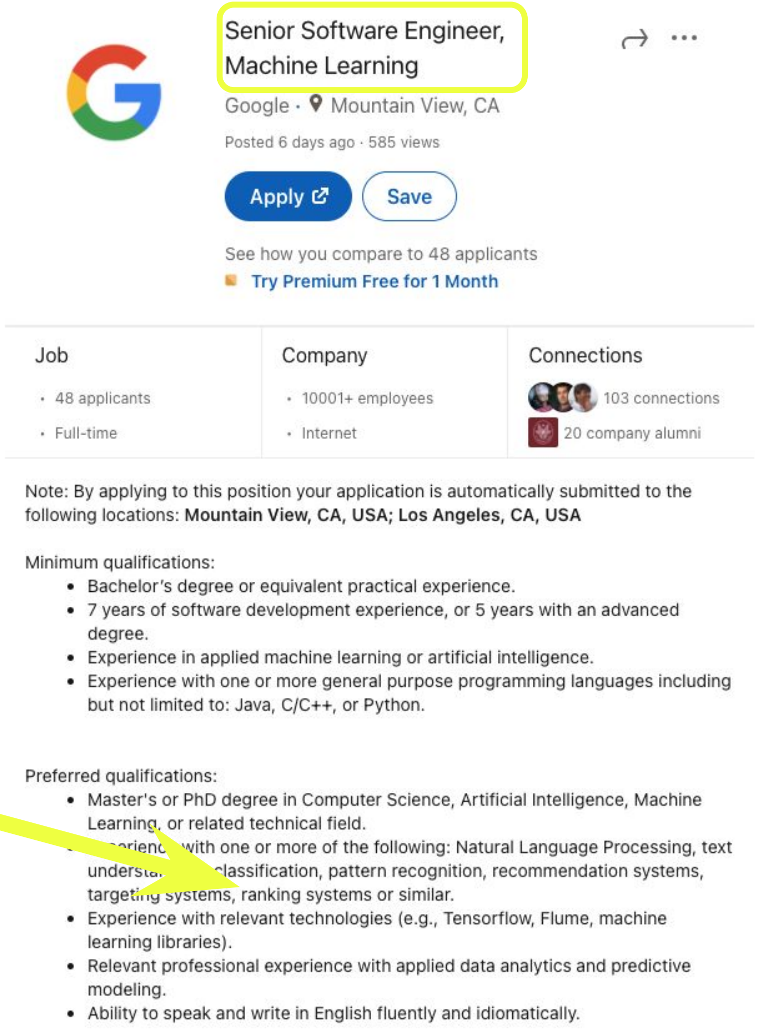


Can we do more?

Machine Learning to Rank Results

- 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
  - Naive Bayes.
- 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”

SAPIENZA
UNIVERSITÀ DI ROMA



Senior Software Engineer, Machine Learning

Google · Mountain View, CA

Posted 6 days ago · 585 views

Apply Save

See how you compare to 48 applicants

Try Premium Free for 1 Month

Job	Company	Connections
48 applicants	10001+ employees	103 connections
Full-time	Internet	20 company alumni

Note: By applying to this position your application is automatically submitted to the following locations: **Mountain View, CA, USA; Los Angeles, CA, USA**

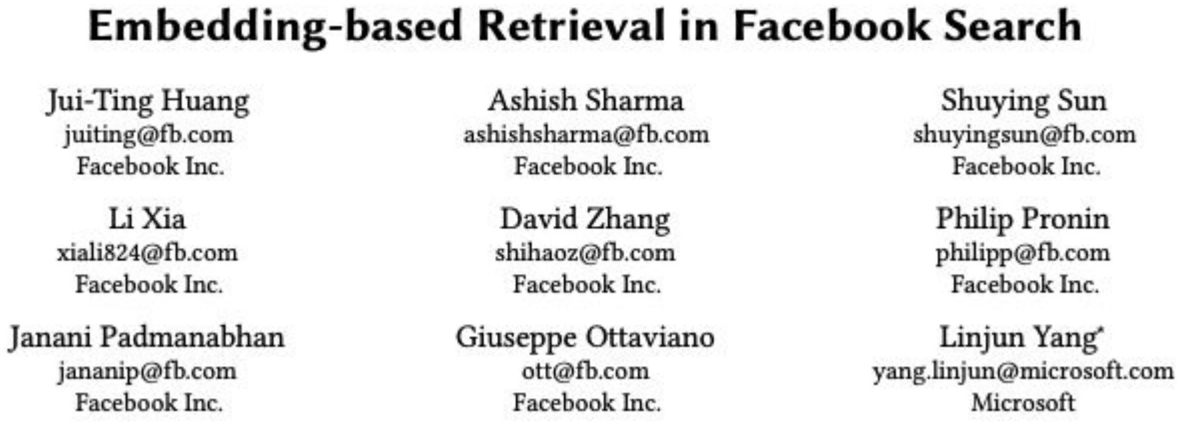
Minimum qualifications:

- Bachelor's degree or equivalent practical experience.
- 7 years of software development experience, or 5 years with an advanced degree.
- Experience in applied machine learning or artificial intelligence.
- Experience with one or more general purpose programming languages including but not limited to: Java, C/C++, or Python.

Preferred qualifications:

- Master's or PhD degree in Computer Science, Artificial Intelligence, Machine Learning, or related technical field.
- erienc with one or more of the following: Natural Language Processing, text understa classification, pattern recognition, recommendation systems, targeting systems, ranking systems or similar.
- Experience with relevant technologies (e.g., Tensorflow, Flume, machine learning libraries).
- Relevant professional experience with applied data analytics and predictive modeling.
- Ability to speak and write in English fluently and idiomatically.

Major Search Engines (even FB's one) use MLR


Embedding-based Retrieval in Facebook Search

Jui-Ting Huang
juiting@fb.com
Facebook Inc.

Ashish Sharma
ashishsharma@fb.com
Facebook Inc.

Shuying Sun
shuyingsun@fb.com
Facebook Inc.

Li Xia
xiali824@fb.com
Facebook Inc.

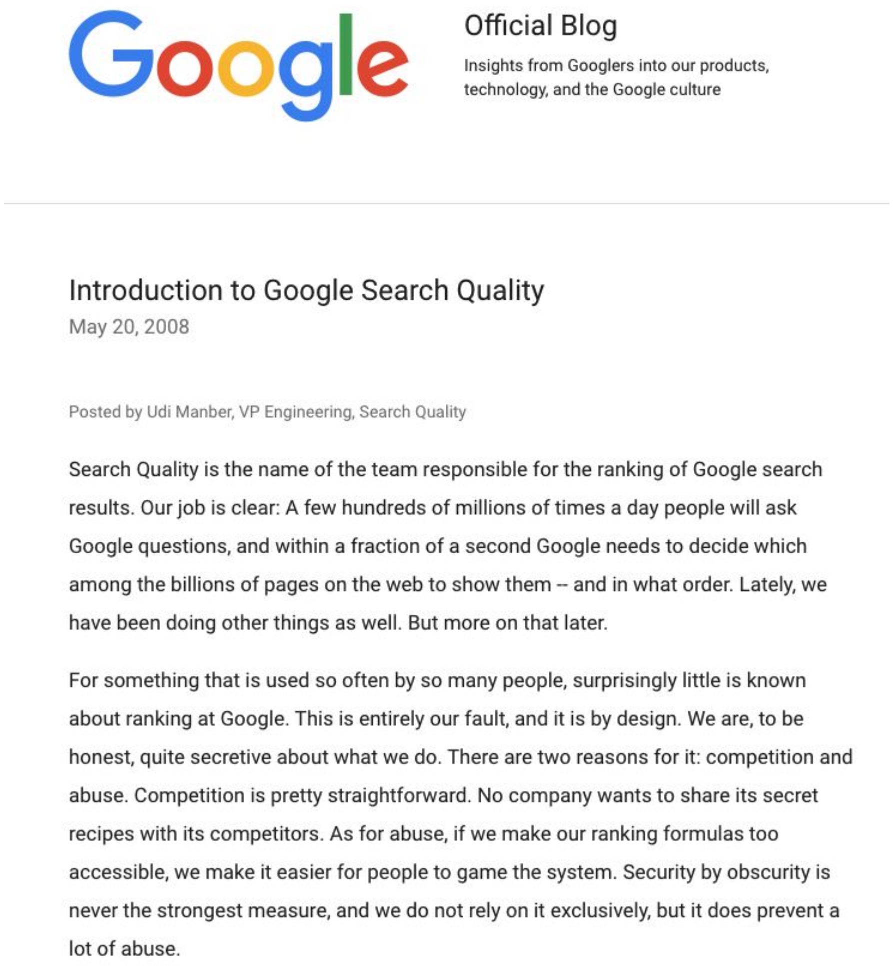
David Zhang
shihaoz@fb.com
Facebook Inc.

Philip Pronin
philipp@fb.com
Facebook Inc.

Janani Padmanabhan
jananip@fb.com
Facebook Inc.

Giuseppe Ottaviano
ott@fb.com
Facebook Inc.

Linjun Yang*
yang.linjun@microsoft.com
Microsoft



Google Official Blog

Insights from Googlers into our products, technology, and the Google culture

Introduction to Google Search Quality

May 20, 2008

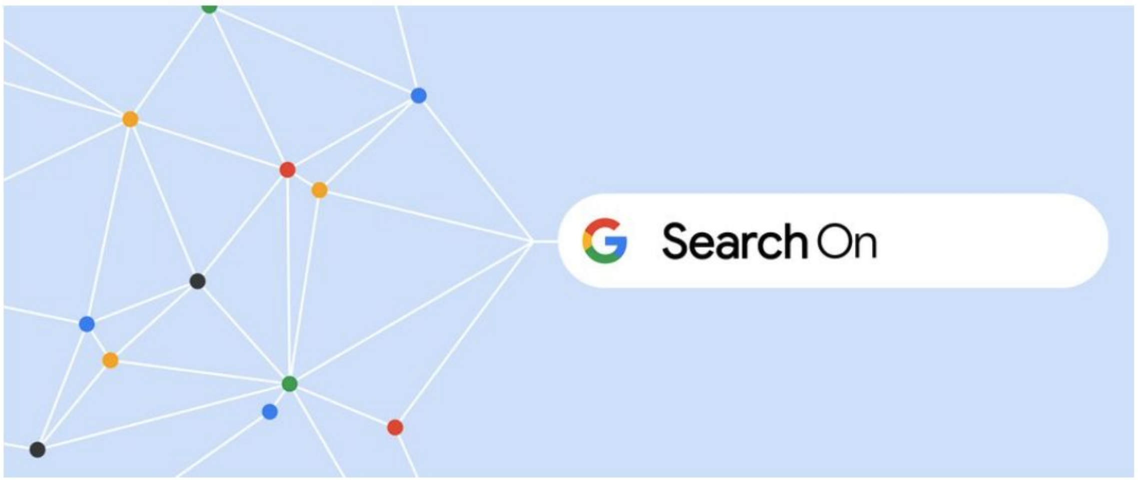
Posted by Udi Manber, VP Engineering, Search Quality

Search Quality is the name of the team responsible for the ranking of Google search results. Our job is clear: A few hundreds of millions of times a day people will ask Google questions, and within a fraction of a second Google needs to decide which among the billions of pages on the web to show them -- and in what order. Lately, we have been doing other things as well. But more on that later.

For something that is used so often by so many people, surprisingly little is known about ranking at Google. This is entirely our fault, and it is by design. We are, to be honest, quite secretive about what we do. There are two reasons for it: competition and abuse. Competition is pretty straightforward. No company wants to share its secret recipes with its competitors. As for abuse, if we make our ranking formulas too accessible, we make it easier for people to game the system. Security by obscurity is never the strongest measure, and we do not rely on it exclusively, but it does prevent a lot of abuse.

https://googleblog.blogspot.com/2008/05/introduction-to-google-search-quality.html

How AI is powering a more helpful Google



Prabhakar Raghavan
Senior Vice President, Search & Assistant, Geo, Ads, Commerce, Payments & NBU

Published Oct 15, 2020

When I first came across the web as a computer scientist in the mid-90s, I was struck by the sheer volume of information online, in contrast with how hard it was to find what you were looking for. It was then that I first started thinking about search, and I've been fascinated by the problem ever since.

We've made tremendous progress over the past 22 years, making Google Search work better for you every day. With recent advancements in AI, we're making bigger leaps forward in improvements to Google than we've seen over the last decade, so it's even easier for you to find just what you're looking for. Today during our Search On livestream, we shared how we're bringing the most advanced AI into our products to further our mission to organize the world's information and make it universally accessible and useful.

https://blog.google/products/search/search-on/

ML for Ranking in IR

- This “good idea” has been actively researched – and actively deployed by major web search engines – in the last 20 years
- Why didn’t it happen earlier?
- Modern supervised ML has been around for about 35 years…
- Naïve Bayes has been around for about 70 years…

SAPIENZA
UNIVERSITÀ DI ROMA

Truth to be told...

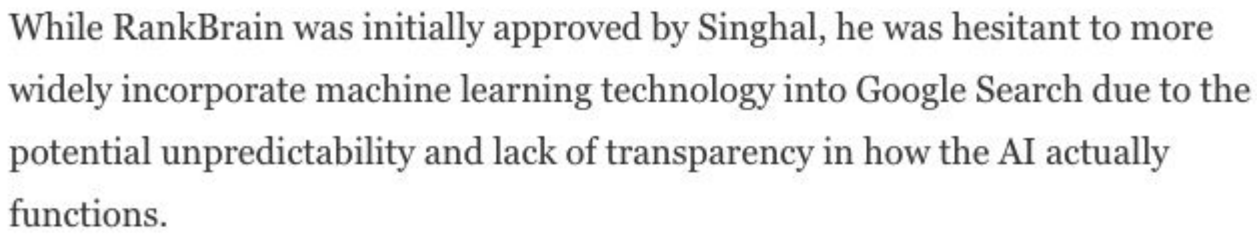
- There’s some truth to the fact that the IR community wasn’t very connected to the ML community
- But there were a whole bunch of precursors:
    - Wong, S.K. et al. 1988. Linear structure in information retrieval. SIGIR 1988.
    - Fuhr, N. 1992. Probabilistic methods in information retrieval. Computer 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 (Google’s Amit Singhal)



While RankBrain was initially approved by Singhal, he was hesitant to more widely incorporate machine learning technology into Google Search due to the potential unpredictability and lack of transparency in how the AI actually functions.

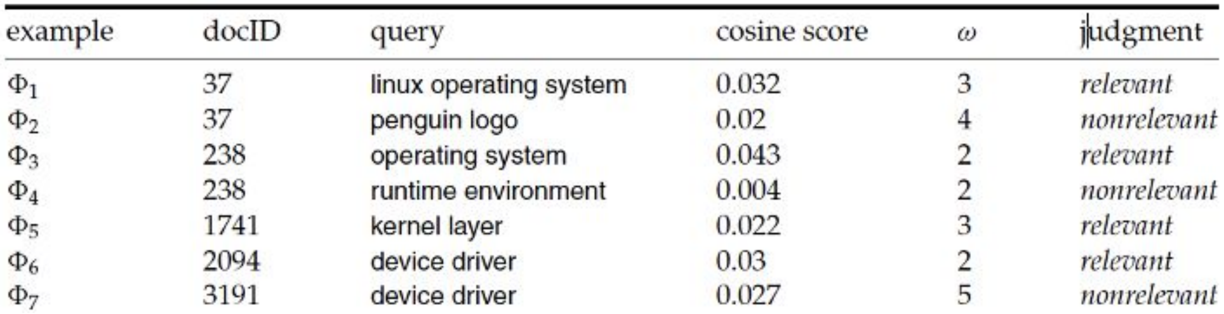
https://www.csmonitor.com/Technology/2016/0204/After-losing-a-pioneering-leader-where-is-Google-s-AI-search-going

… Nowadays

- 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: 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
    - $x = (\alpha, \omega) \rightarrow \alpha$ 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



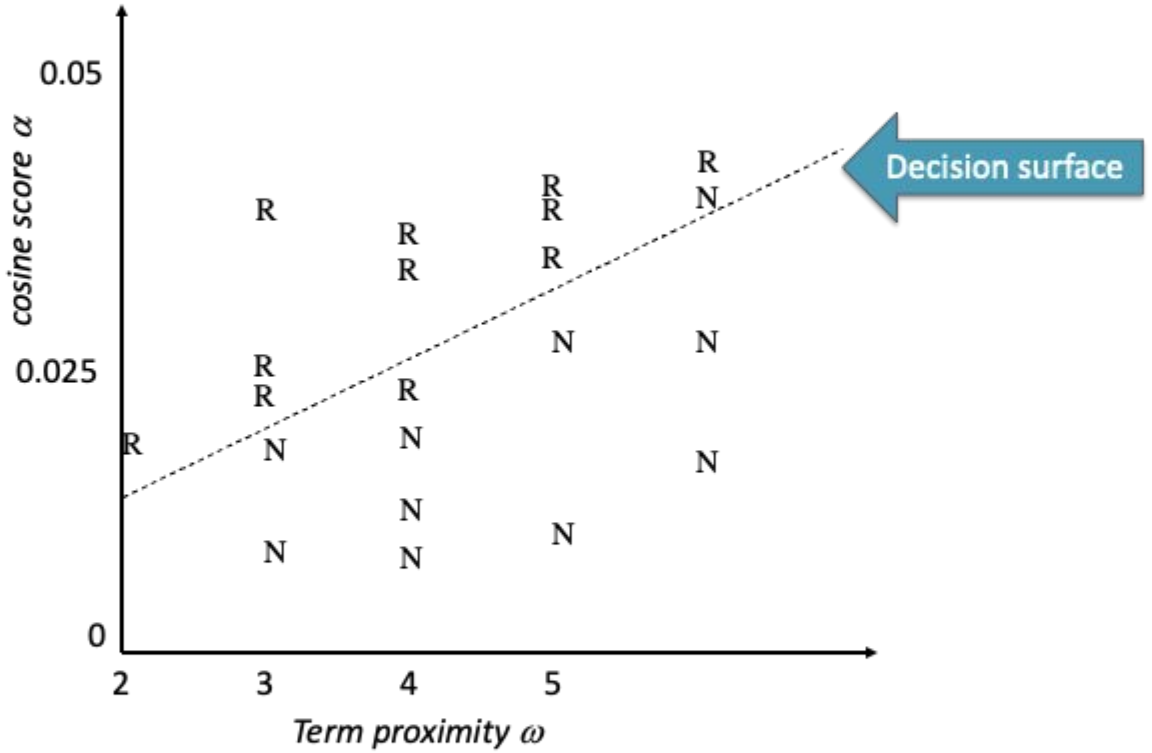
example	docID	query	cosine score	ω	judgment
Φ_1	37	linux operating system	0.032	3	*relevant*
Φ_2	37	penguin logo	0.02	4	*nonrelevant*
Φ_3	238	operating system	0.043	2	*relevant*
Φ_4	238	runtime environment	0.004	2	*nonrelevant*
Φ_5	1741	kernel layer	0.022	3	*relevant*
Φ_6	2094	device driver	0.03	2	*relevant*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

Simple example: 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: Classification for ad hoc IR



Using classification for search ranking

- 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

- Let relevance score $g(r|d,q) = wf(d,q) + b$
- 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

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

Is it really 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
  - Ordinal regression (or “ranking”) problems: Map to an ordered set of classes

- 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: $c_1 < c_2 < \dots < c_J$
  - 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

Learning to Rank in Search

- Support Vector Machines (Vapnik, 1995)
  - Adapted to ranking: Ranking SVM (Joachims 2002)
- Neural Nets:
  - RankNet (Burges et al., 2006)
  - Unified Embedding Framework (Huang et al., 2020)
- 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, Google, …
- 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

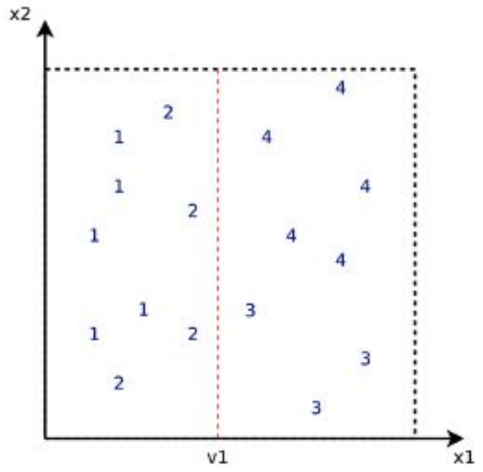
Regression Trees

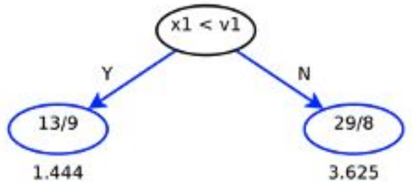
Regression trees

- 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
  - $\gamma_k = f(x_i) =$ AVG(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 Regression trees

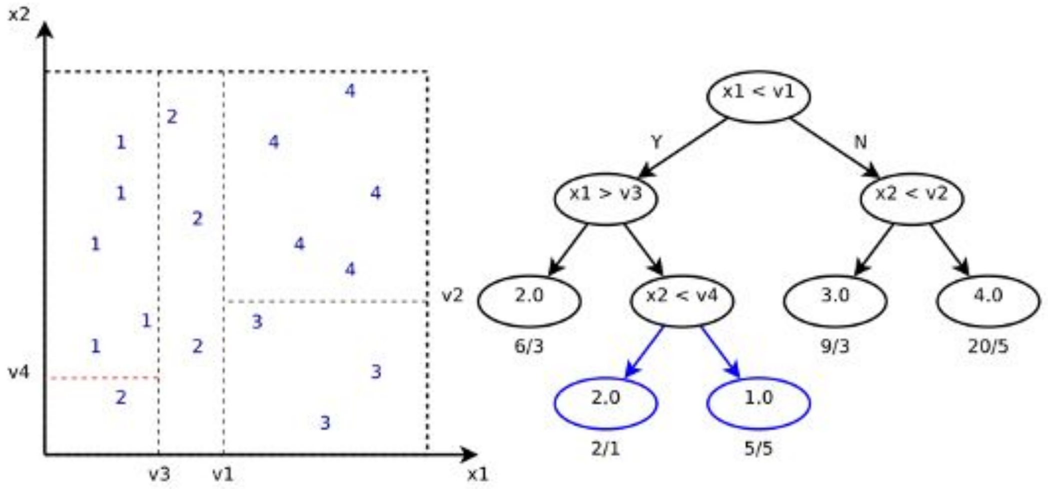
- The algorithm searches for split variables and split points, x1 and v1 so as to minimize the predicted error, i.e., $\sum_i (y_i - f(x_i))^2$





Training Regression trees

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



What’s boosting, anyway?

- 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

Gradient Boosting

- 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:
  - $F^*(\boldsymbol{x})=\text{argmin}_{F(\boldsymbol{x})}\ \mathbb{E}_{(y,\boldsymbol{x})}L(y,\ F(\boldsymbol{x}))$
- Boosting approximates $F^*(\boldsymbol{x})$ by an additive expansion
  - $F(\boldsymbol{x})=\sum\ [\beta_m\ h(\boldsymbol{x};\boldsymbol{a}_m)]$ for m in [1, M]
- where h(**x**; a) are simple functions of **x** with parameters a = {a_1, a_2, …, a_n} defining the function h, and the β are weighting coefficients

Fitting Parameters

- Function parameters are iteratively fit to the training data:
  - Set $F_0(x)$ = initial guess (or zero)
  - For each m = 1, 2, …, M
    - $\boldsymbol{a}_m$ = $\text{argmin}_{\boldsymbol{a}} \sum_i L(y_i, F_{(m-1)}(\boldsymbol{x}_i) + \beta h(\boldsymbol{x}_i, \boldsymbol{a}))$
    - $F_m(\boldsymbol{x}) = F_{(m-1)}(\boldsymbol{x}_i) + \beta h(\boldsymbol{x}_i, \boldsymbol{a}_\text{m})$

- You successively estimate and add a new tree to the sum
- You never go back to revisit past decisions

Fitting Parameters

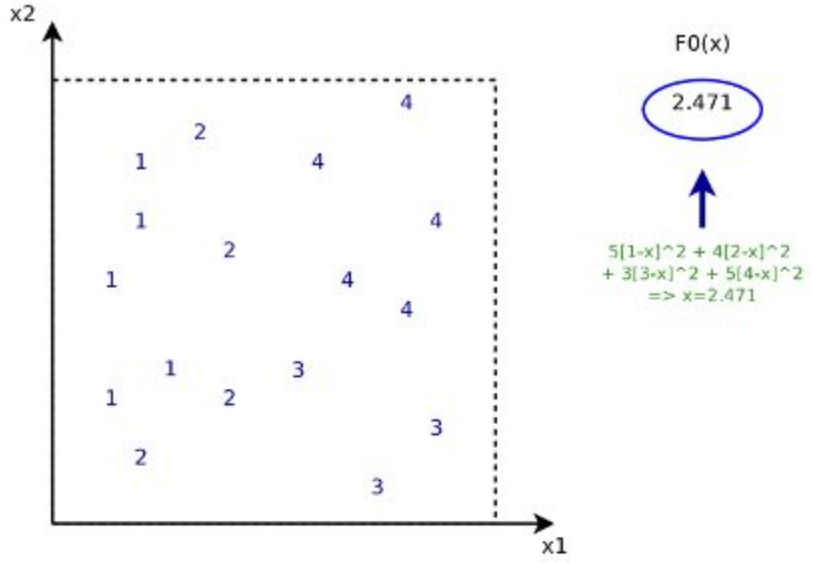
- Gradient boosting approximately achieves this for any differentiable loss function
  - Fit the function h(x; a) by least squares
    - $a_m = \text{argmin}_a \sum_i [(\tilde{y}_{im} - h(x_i, a)]^2$
  - to the "pseudo-residuals" (deviation from desired scores)
    - $\tilde{y}_{im} = -\left[\frac{\partial L(y_i, F(\mathbf{x}_i))}{\partial F(\mathbf{x}_i)}\right]_{F(\mathbf{x})=F_{m-1}(\mathbf{x})}$
- 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(x; 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.?)

Gradient Tree Boosting: Learning

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



Gradient Tree Boosting: Learning

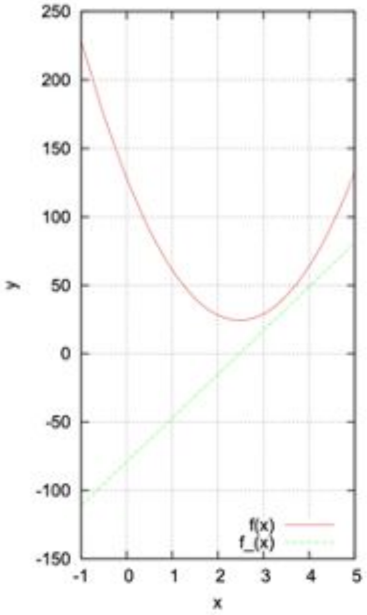
- We want to find value γ_{km} for root node of tree

Quadratic loss for the leaf (red):

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

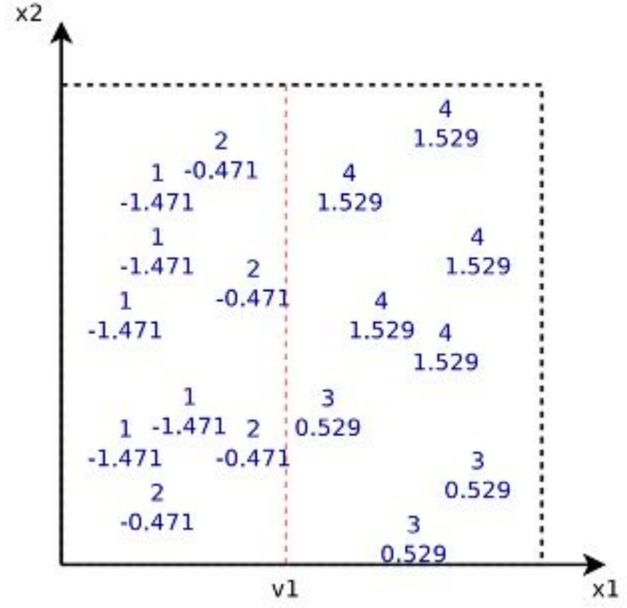
$f(x)$ is quadratic, *convex*
$\Rightarrow$ Optimum at $f'(x) = 0$ (green)

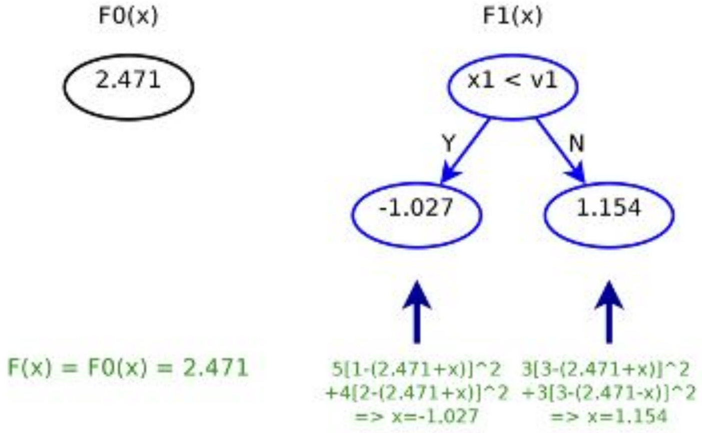
$$\frac{\partial f(x)}{\partial x} = 5 \cdot (-2+2x) + 4 \cdot (-4+2x)^2 + 3 \cdot (-6+2x)^2 + 5 \cdot (-8+2x)^2 = -84 + 34x = 34(x - 2.471)$$



Gradient Tree Boosting: Learning

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

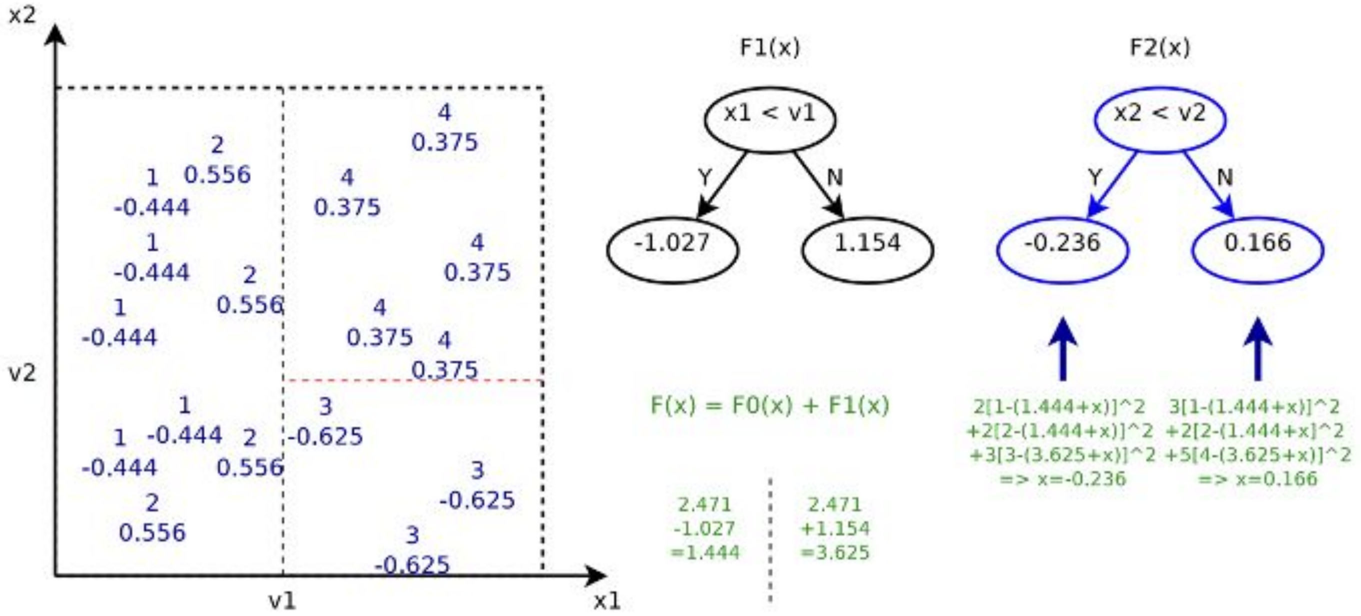




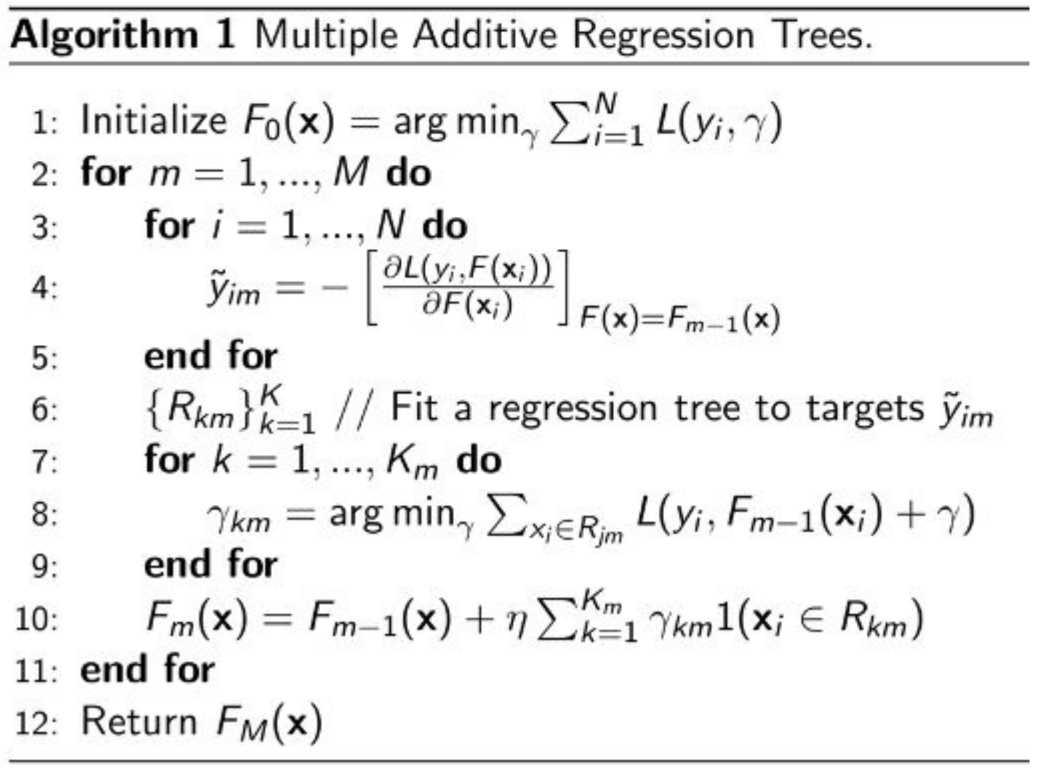
F0(x)
2.471
F1(x)
x1 < v1
Y
N
-1.027
1.154
F(x) = F0(x) = 2.471
5[1-(2.471+x)]^2 +4[2-(2.471+x)]^2 => x=-1.027
3[3-(2.471+x)]^2 +3[3-(2.471-x)]^2 => x=1.154

Gradient Tree Boosting: Learning

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



Multiple Additive Regression Trees (MART)



Algorithm 1 Multiple Additive Regression Trees.

```
1:  Initialize $F_0(\mathbf{x}) = \arg\min_\gamma \sum_{i=1}^{N} L(y_i, \gamma)$
2:  for $m = 1, ..., M$ do
3:      for $i = 1, ..., N$ do
4:          $\tilde{y}_{im} = -\left[\frac{\partial L(y_i, F(\mathbf{x}_i))}{\partial F(\mathbf{x}_i)}\right]_{F(\mathbf{x})=F_{m-1}(\mathbf{x})}$
5:      end for
6:      $\{R_{km}\}_{k=1}^{K}$ // Fit a regression tree to targets $\tilde{y}_{im}$
7:      for $k = 1, ..., K_m$ do
8:          $\gamma_{km} = \arg\min_\gamma \sum_{x_i \in R_{jm}} L(y_i, F_{m-1}(\mathbf{x}_i) + \gamma)$
9:      end for
10:     $F_m(\mathbf{x}) = F_{m-1}(\mathbf{x}) + \eta \sum_{k=1}^{K_m} \gamma_{km} 1(\mathbf{x}_i \in R_{km})$
11: end for
12: Return $F_M(\mathbf{x})$
```

Historical Path to LambdaMART via RankNet (NN)

- 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 $d_i > d_j$ via:
  - $P_{ij} = P(d_i > d_j) = 1/(1+e^{(-\sigma(si-sj))})$
- 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; $\bar{P}_{ij}$ the actual probability (0 or 1 for categorical judgments)

Historical Path to LambdaMART via RankNet (NN)

- Combining these equations gives
- $C = 1/2(1-S_{ij})\sigma(s_i - s_j) + \log(1+e^{(-\sigma(si-sj))})$
- 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

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

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

SAPIENZA
UNIVERSITÀ DI ROMA

Historical Path to LambdaMART via RankNet (NN)

- 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

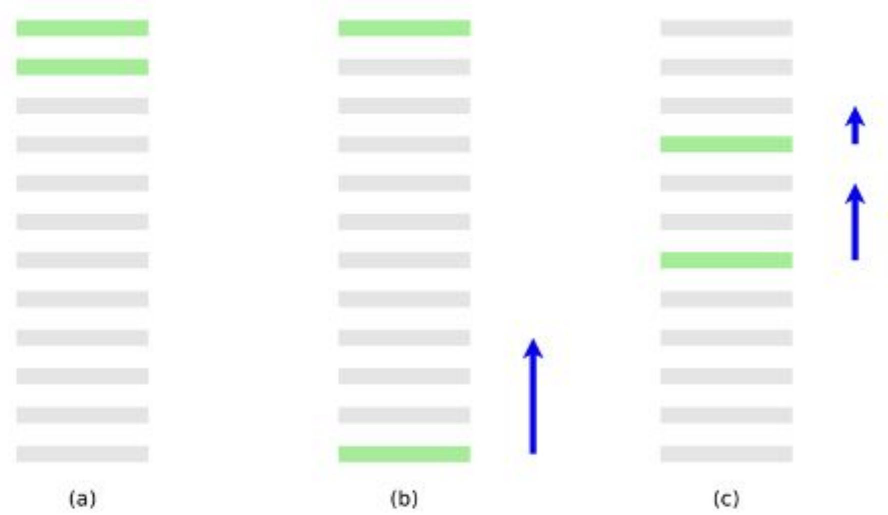
$$\lambda_i = \sum_{j:\{i,j\}\in I} \lambda_{ij} - \sum_{k:\{k,i\}\in I} \lambda_{ki}$$

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

SAPIENZA
UNIVERSITÀ DI ROMA

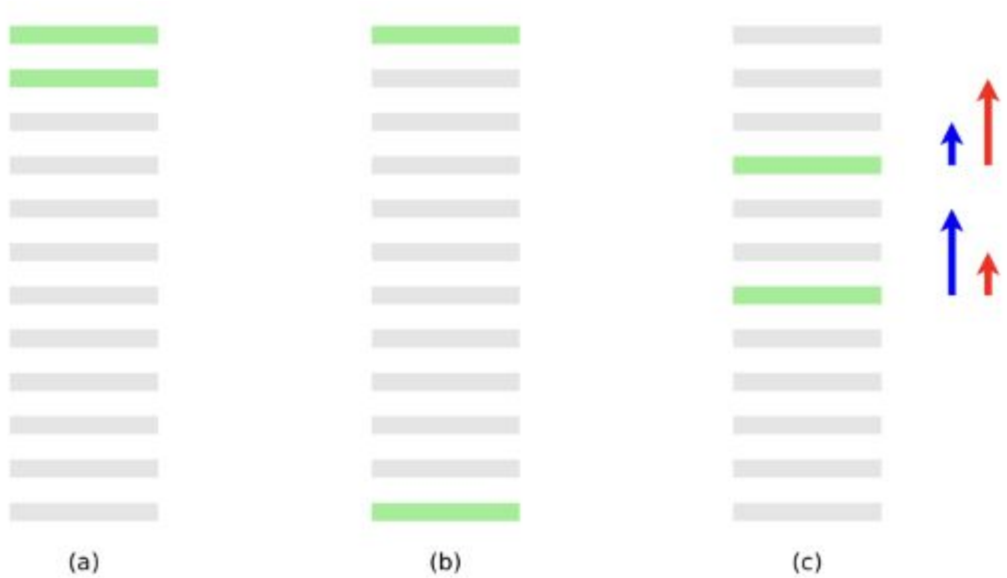
RankNet lambdas

- (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

- 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 RankNet to LambdaRank

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

LambdaRank Algorithm

```
**set** number of trees $N$, number of training samples $m$, number of leaves per tree $L$,
learning rate $\eta$
**for** $i = 0$ to $m$ **do**
    $F_0(x_i) = \text{BaseModel}(x_i)$    //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 $\eta$.
**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

Takeaway Messages

- The idea of learning ranking functions has been around for about 30 years
- ML knowledge, availability of training datasets, a rich space of features, and massive computation came together to make this a hot research area
- Typically LambdaMART outperforms other methods in real search ranking tasks
- MLR over many features now easily beats traditional hand-designed ranking functions in comparative evaluations
    - [in part by using the hand-designed functions as features!]
- Next Step: NeuralIR!

SAPIENZA
UNIVERSITÀ DI ROMA