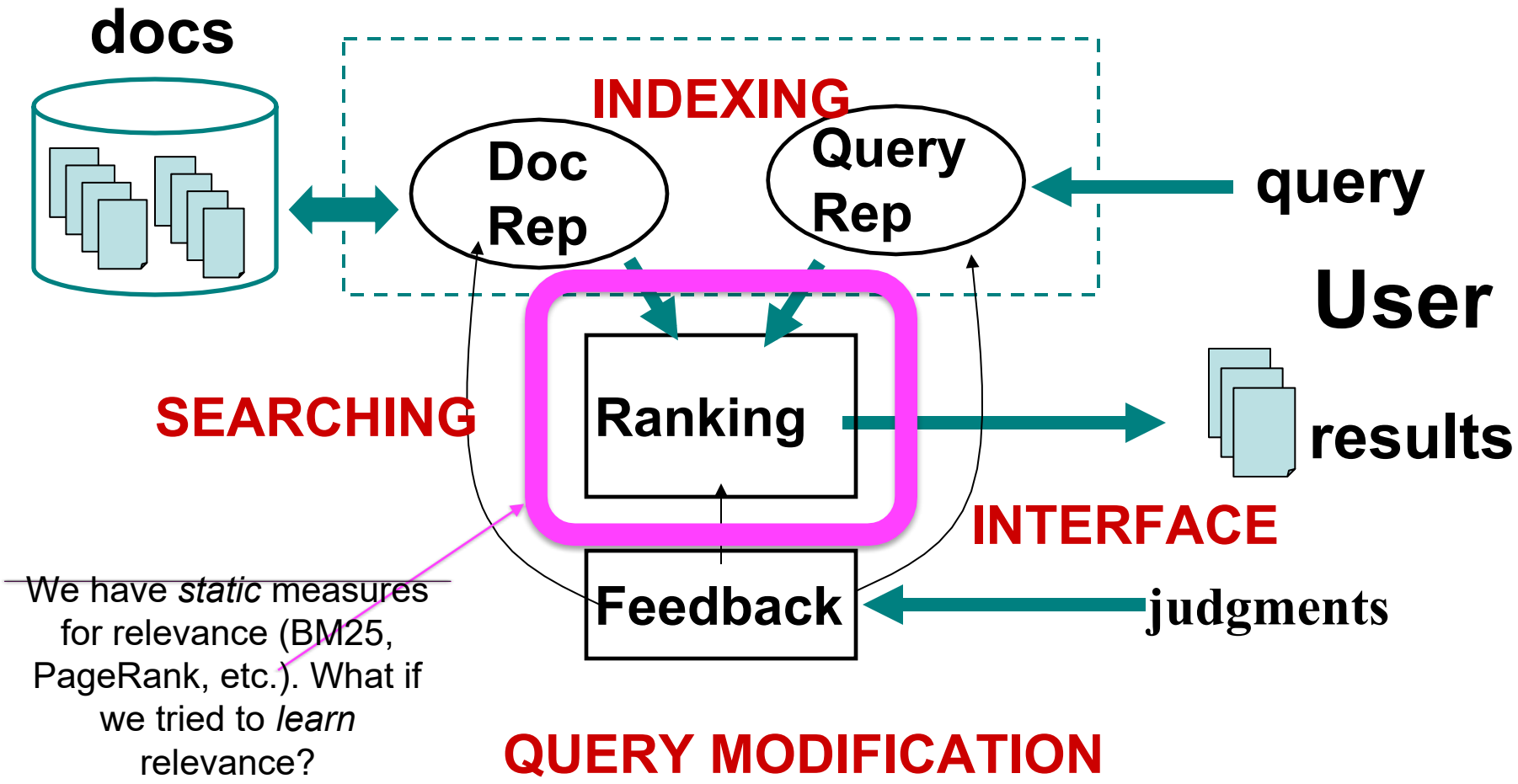
Learning To Rank

Lecture for SI650 / EECS 549
Information Retrieval

October 1, 2025

Some slides adapted from Chris Manning and Ambuj Tewari,

A Typical IR System Architecture



Learning to Rank

Many search collections have rich information

- We've looked at methods for ranking documents in IR
  - Cosine similarity, inverse document frequency, proximity, pivoted document length normalization, language models, BM25 …
- Documents (and document, query pairs) include rich information we haven't relied on yet
  - Think of your HW and the Wikipedia dataset and imagine you were building a search engine for that. What would you use?

- Surely we can also use machine learning to rank the documents displayed in search results?
  - Sounds like a good idea
  - A.k.a. "machine-learned relevance" or "learning to rank"

Machine learning for IR ranking

- This “good idea” has been actively researched – and actively deployed by major web search engines – in the last 10 or so years
- Why didn’t it happen earlier?
  - Modern supervised ML has been around for about 20 years…
  - Naïve Bayes has been around for about 50 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 to tune weighting coefficients by hand
  - And people did

Why is ML needed now?

- Modern systems—especially on the Web—use a great number of features:
  - Arbitrary useful features – not a single unified model
  - 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 length?
- The New York Times (2008-06-03) quoted Amit Singhal as saying Google was using over 200 such features.

In a ML setting, features are joint query-document features

- Number of query terms that occur in document
- Document length
- Sum/min/max/mean/variance of term frequencies
- Sum/min/max/mean/variance of tf-idf
- Cosine similarity between query and document
- Any model of P(R = 1|Q, D) gives a feature (e.g., BM25)
- No. of slashes in/length of URL
- Pagerank,
- site level Pagerank
- Query-URL click count
- user-URL click count

Typical Learning to Rank Loop

- Feature construction: Find a way to map (query , webpage) into $\mathbb{R}_p$
  - Each example (query, *m* webpages) gets mapped to $X_n = \mathbb{R}^{m \times p}$
- Obtain relevance labels: Get $R_n \in \{0, 1, 2, \ldots, R_{max}\}^m$ from human judgements
- Train ranker: Learn a ranking function $f : X_n \rightarrow S_m$ where S_m is the set of m-permutations
  - A permutation is a particular ordering of the results
  - Training usually done by solving some optimization problems on the training set
- Evaluate performance: Using some performance measure, evaluate the ranker on a “test set”

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)
  - Document 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 a **very 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

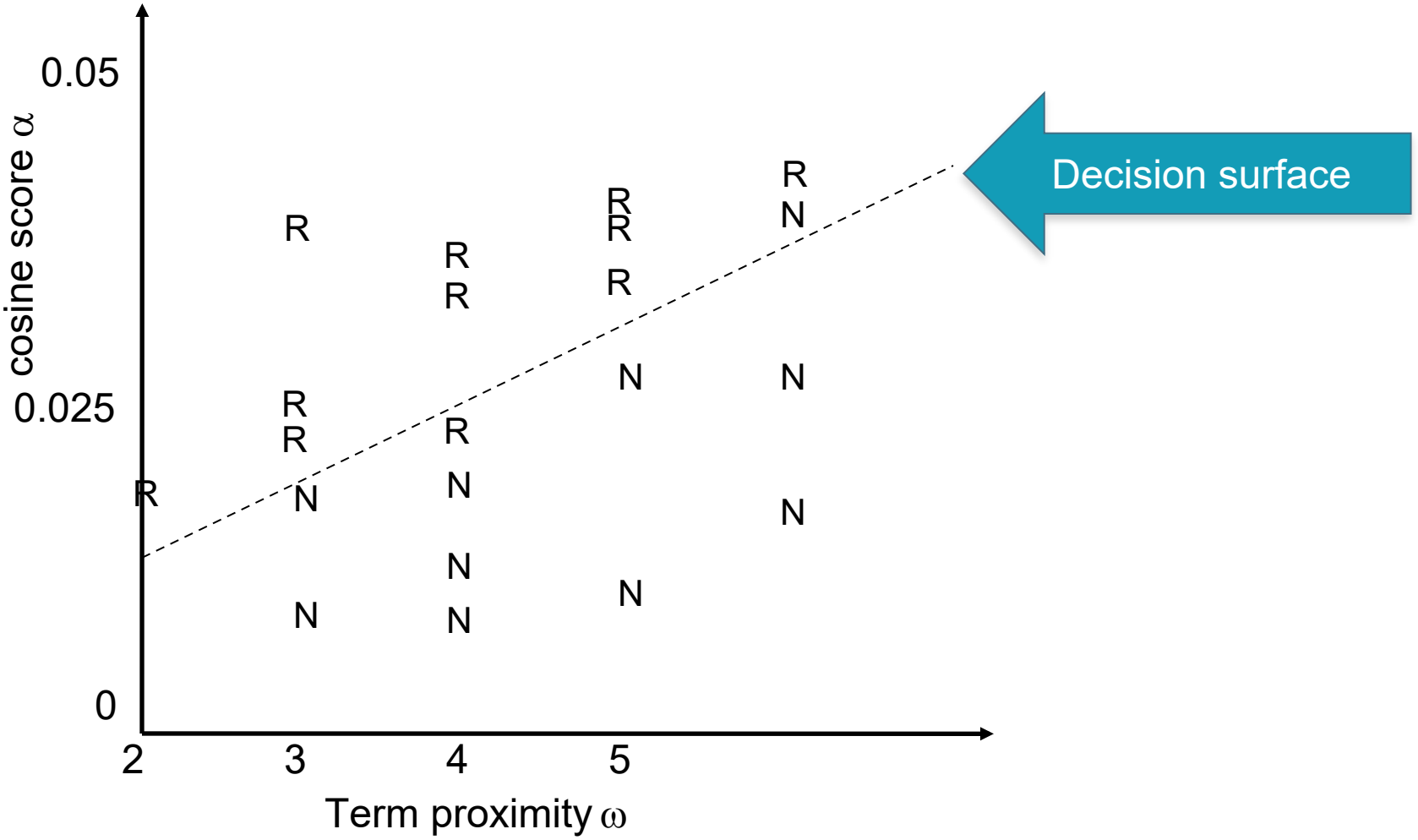
  $$\text{Score}(d, q) = \text{Score}(\alpha, \omega) = a\alpha + b\omega + c$$

- And the linear classifier is

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

- … just like when we were doing text classification

Simple example: Using classification for ad hoc IR



第 1 页：问题设定与特征

- 训练样本是 **(q, d, r)** 三元组，r 为二元（也可扩展为多级）。
- 为每个 *(q,d)* 提取两个特征：
  - **α**：查询与文档的**余弦相似度**；
  - **ω**：**最小覆盖窗口**（把所有查询词都包含在内的最短文本跨度，越小表示词越"靠近"）。
- 强调"**查询词近邻/紧密度**"是很关键的新加权因素（传统方法里常被忽视）。

SI650-Week-06-Learning to Rank

第 2 页：线性打分与阈值判定

- 定义线性打分：**Score(d,q) = a•α + b•ω + c**。
- 给定阈值 **θ**，若 **Score(d,q) > θ** 则判为**相关**，否则**不相关**。
- 这和我们做文本分类时用线性分类器几乎一模一样，只是这里的输入是 *(q,d)* 的联合特征。

SI650-Week-06-Learning to Rank

第 3 页：二维直观图与要点

- 在"**余弦相似度 α–词距 ω**"平面上画散点：相关样本一般出现在"α 高、ω 小"的区域；
- 一条近似线性**决策边界**即可把相关与不相关分开，支持上面的线性判别做法；
- 延伸启示：可在此框架上加入更多特征、用更强分类器（SVM/神经网络等）以提升排序效果。

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

- We can generalize this to classifier functions over more features
- We can use methods we have seen previously for learning the linear classifier weights

An SVM classifier for information retrieval [Nallapati 2004]

- Let $g(r|d,q) = \mathbf{w} \cdot f(d,q) + b$
- SVM training: 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 (why?) 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 that uses a Language Model (LM) based IR
  - 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)
Disk 3	LM	**0.1785**	**0.2503**	0.2666
	SVM	0.1728	0.2432	**0.2750**
Disk 4-5	LM	**0.1773**	**0.2516**	0.2656
	SVM	0.1646	0.2355	**0.2675**

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

SVM（Support Vector Machine，支持向量机）是一种**监督学习**方法，主要用于**二分类**（也可扩展到回归与多分类）。它的核心思想是：

在特征空间里找一条（或一个）**最大间隔的分离超平面**，让不同类别的数据被分开，并且间隔（margin）尽可能大——这样对噪声更鲁棒、泛化更好。

直觉

- 你可以把 SVM 想成"在高维空间画一条最稳妥的分界线"，只用**离边界最近的少量样本**（叫**支持向量**）来决定这条线的位置；其他样本不影响最终边界。

数学骨架（线性可分的简化版）

- 学习一个超平面：$w^\top x + b = 0$
- 目标：最大化几何间隔，相当于最小化 $\frac{1}{2}\|w\|^2$，同时满足 $y_i(w^\top x_i + b) \geq 1$。
- 线性不可分时，用**软间隔**与**惩罚系数** C：允许少量误分类，用更大的 C 惩罚错误更重。

“Learning to rank”

- Classification probably isn’t the right way to think about approaching ad hoc IR:
  - Classification problems: Map to a unordered set of classes
  - Regression problems: Map to a real value
  - Ordinal regression problems: Map to an ordered set of classes
    - A fairly obscure sub-branch of statistics, but closer to what we want here
    - Relations between relevance levels are modeled
    - Documents are not just good, but better/worse versus other documents for 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 represented as feature vectors φ_i and relevance ranking c_i
- We could do point-wise learning, where we try to map items of a certain relevance rank to a subinterval
- But most work does pair-wise learning, where the input is a pair of results for a query, and the class is the relevance ordering relationship between them

Learning to Rank

- A broad family of approaches
- Pointwise (e.g., logistic regression)
  - Input: single documents
  - Output: scores or class labels
- Pairwise (e.g., RankSVM)
  - Input: document pairs
  - Output: partial order preference
- Listwise (e.g., LambdaRank)
  - Input: document collection
  - Output: ranked list of documents

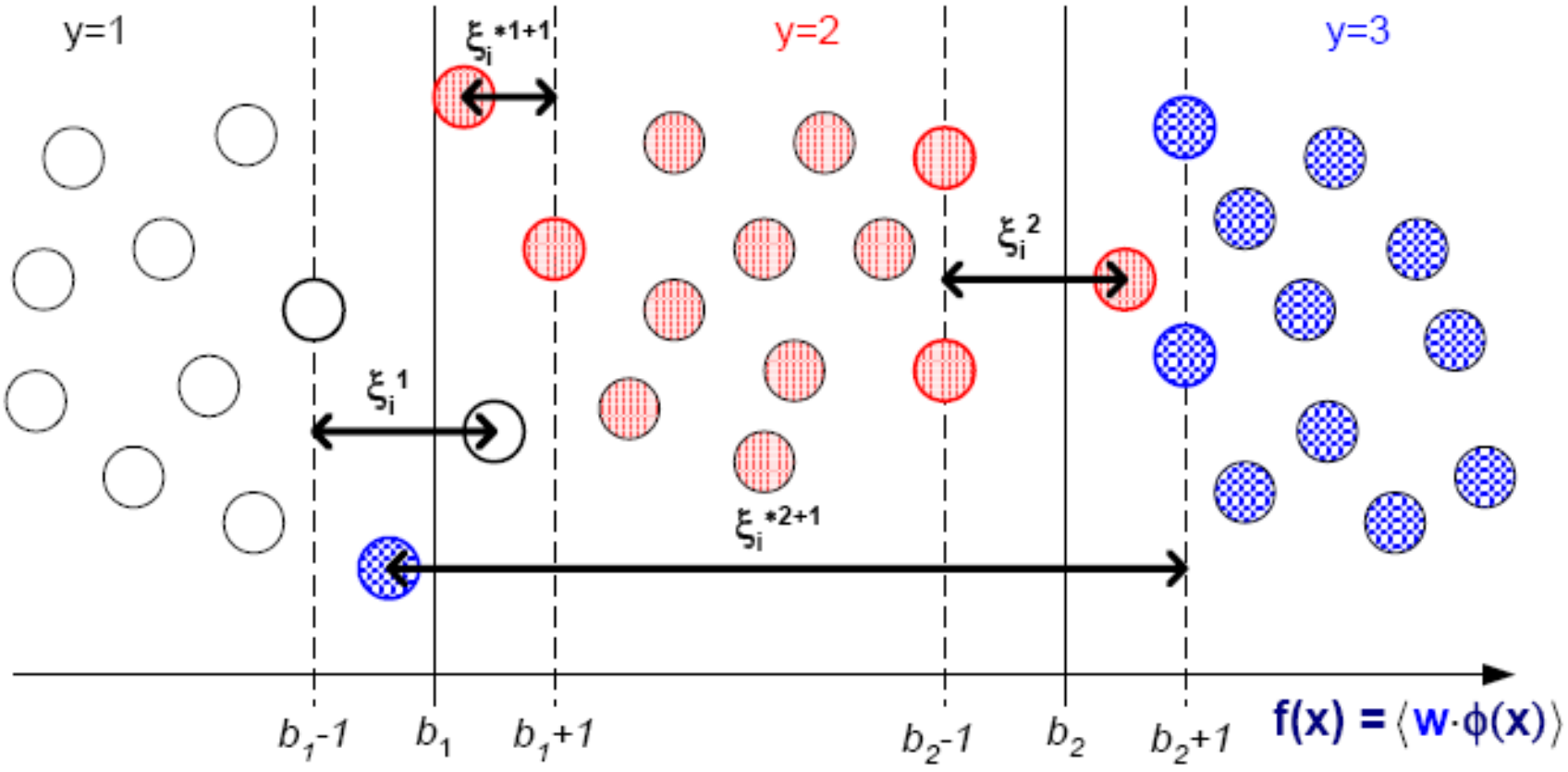
Pointwise ranking

Simple Approach: Linear Regression

- Perhaps the simplest approach: regress the relevance score on query-document features
- Feed examples of the form ($\varphi(q, d_i)$, R_i) to your favorite regression algorithm
  - $\varphi(q, d_i)$ is the function that maps a query and document to a feature vector
  - R_i is the relevance of document *i*
- Advantage: can use already built regression tools
- Disadvantage: there's no real reason for us to predict relevance scores, we're just building a ranking function

Point-wise learning with SVMs

- Goal is to learn a threshold to separate each



Pair-wise Ranking

Pairwise approach

- Training samples: document pairs
- Learning task: classification of object pairs into 2 categories (correctly ranked or incorrectly ranked)
- Methods:
  - RankSVM (Herbrich et al., 1999)
  - RankBoost (Freund et al., 1998)
  - RankNet (Burges et al., 2005)

Pairwise learning: The Ranking SVM (old school)

[Herbrich et al. 1999, 2000; Joachims et al. 2002]

- Aim is to classify instance pairs as correctly ranked or incorrectly ranked
  - This turns an ordinal regression problem back into a binary classification problem
- We want a ranking function f such that

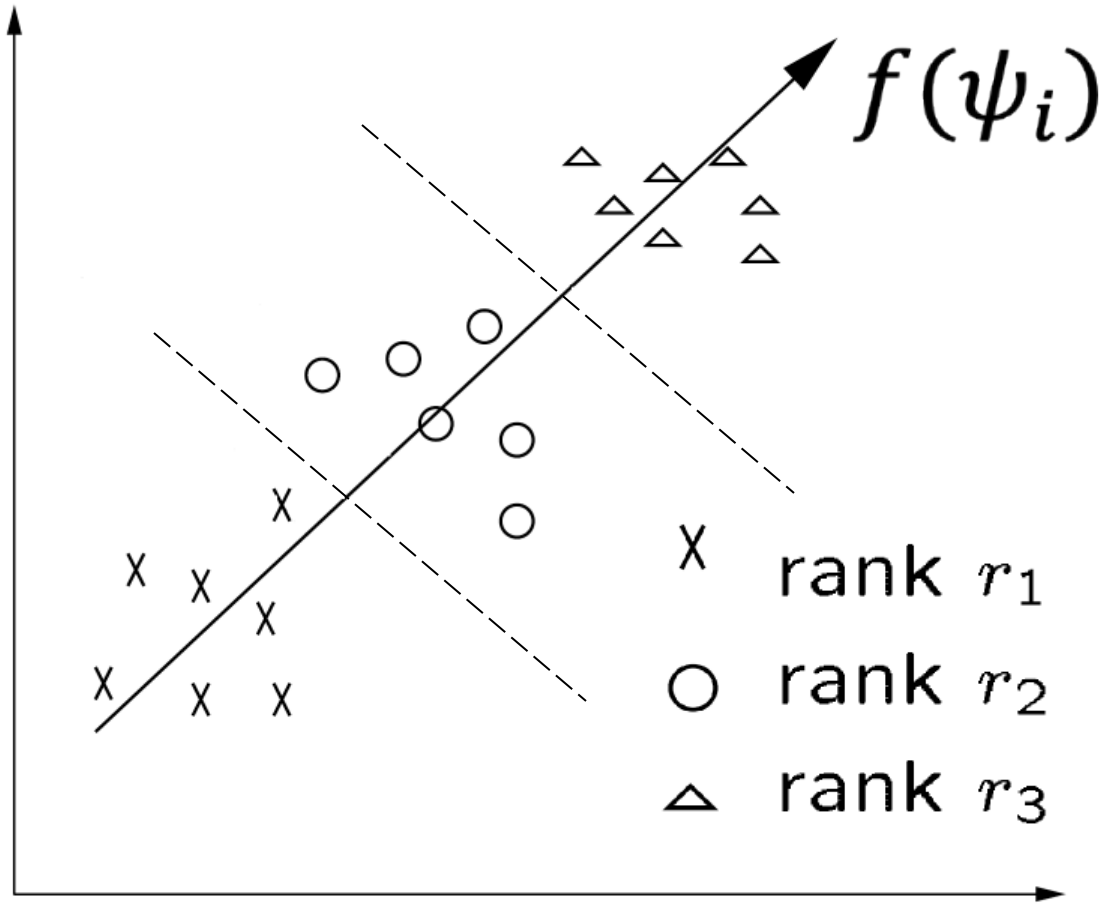
  $c_i > c_k$ iff $f(\psi_i) > f(\psi_k)$
  - … or at least one that tries to do this with minimal error
- Suppose that f is a linear function

  $f(\psi_i) = \mathbf{w} \bullet \psi_i$

The Ranking SVM

[Herbrich et al. 1999, 2000; Joachims et al. 2002]

- Ranking Model: $f(\psi_i)$



The Ranking SVM

[Herbrich et al. 1999, 2000; Joachims et al. 2002]

- Then (combining the two equations on the last slide):

  $c_i > c_k$ iff $\mathbf{w} \bullet (\psi_i - \psi_k) > 0$

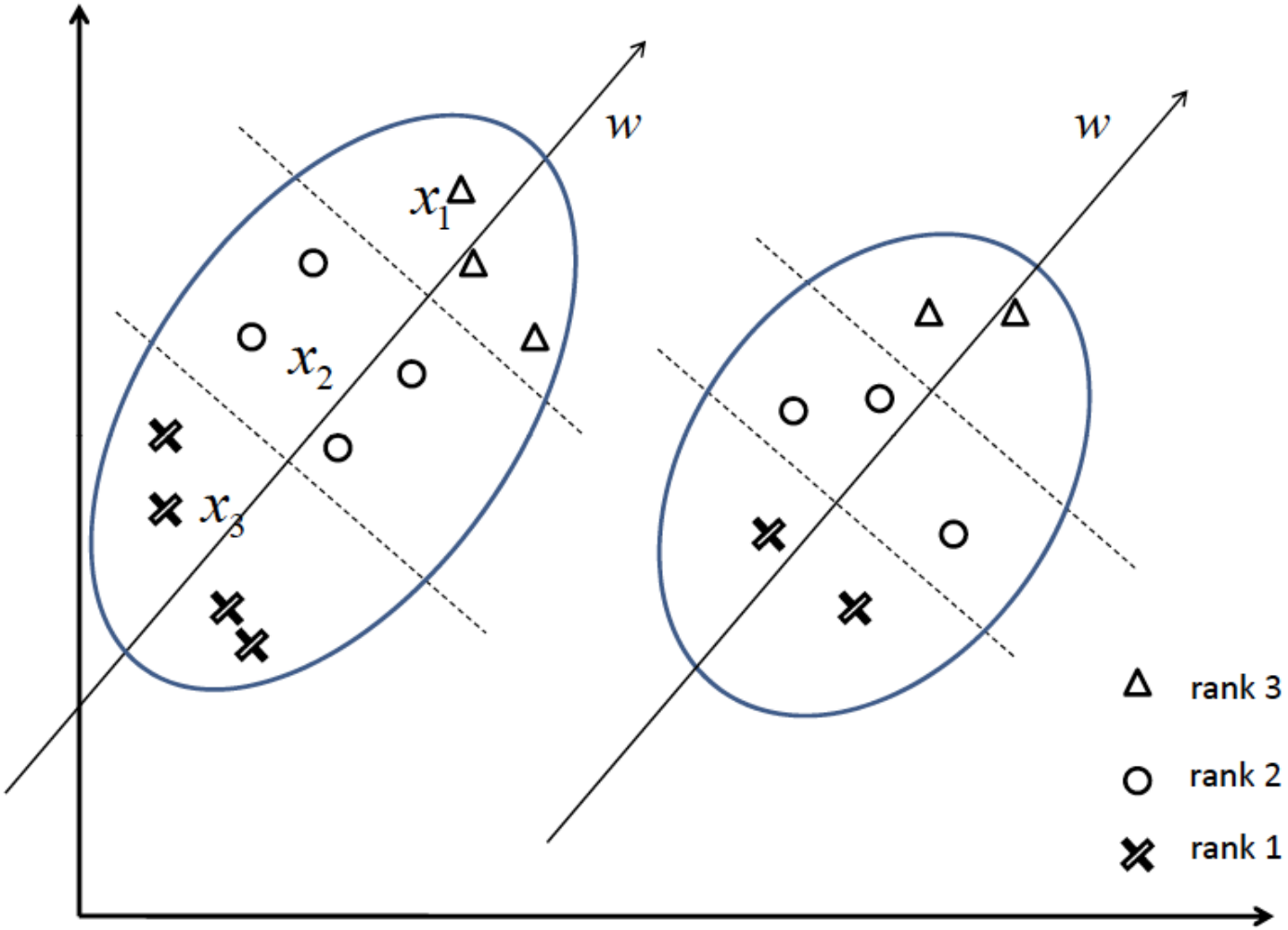
- Let us then create a new instance space from such pairs:

  $\Phi_u = \Phi(d_i, d_j, q) = \psi_i - \psi_k$

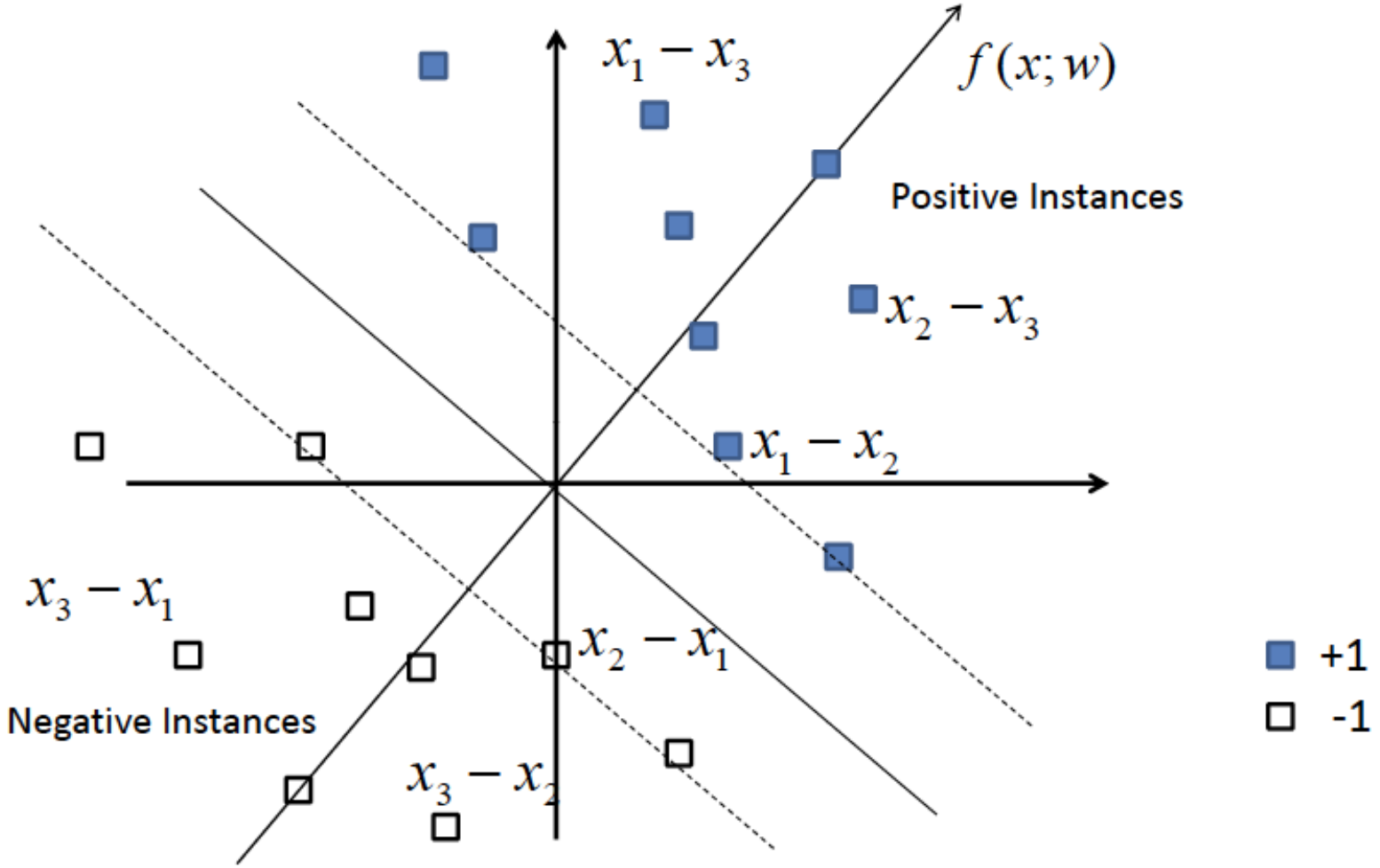
  $z_u = +1, 0, -1$ as $c_i >,=,< c_k$

- We can build model over just cases for which $z_u = -1$
- From training data $S = \{\Phi_u\}$, we train an SVM

Two queries in the original space



Two queries in the pairwise space



The Ranking SVM

[Herbrich et al. 1999, 2000; Joachims et al. 2002]

- The SVM learning task is then like other examples that we saw before
- Exactly the same as binary SVM but with training examples (Φ_u, z_u)instead of (x_i, y_i) for every pairwise difference vector Φ_u.
- But again, if you already have (doc_i,doc_k) with label +1, you don't need (doc_k,doc_i) with label –1 because it's redundant.
  - You can use half the data (pairs)
- You can use sklearn (or other good SVM libraries) to train a RankSVM model!

The ranking SVM fails to model the IR problem well …

- Correctly ordering the most relevant documents is crucial to the success of an IR system, while misordering less relevant results matters little
  - The ranking SVM considers all ordering violations as the same
- Some queries have many (somewhat) relevant documents, and other queries few. If we treat all pairs of results for a query equally, queries with many results will dominate the learning
  - But actually queries with few relevant results are at least as important to do well on

Two Problems with Direct Application of the Ranking SVM

- Cost sensitiveness: negative effects of making errors on top ranked documents

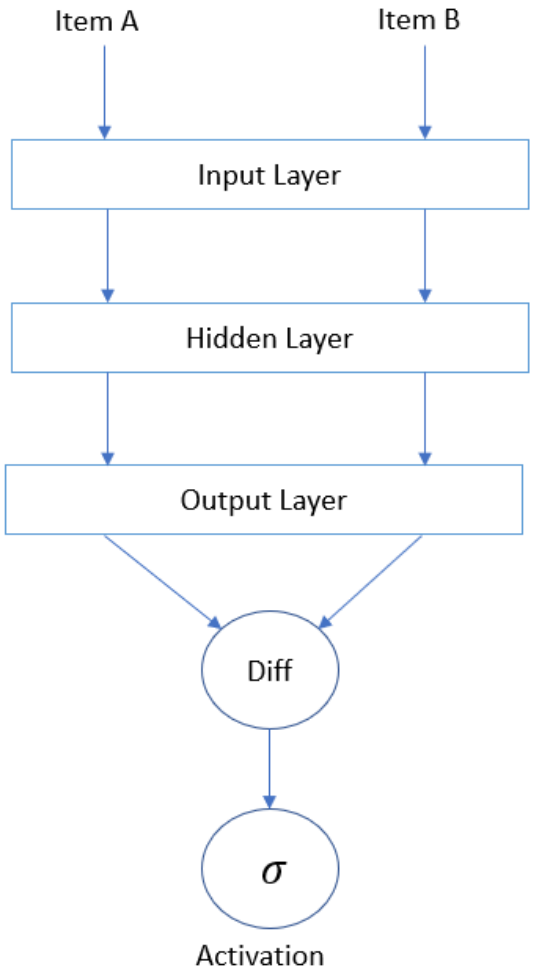
  d: *definitely relevant*, p: *partially relevant*, n: *not relevant*
  ranking 1: p d p n n n n
  ranking 2: d p n p n n n

- Query normalization: number of instance pairs varies according to query

  q_1: d p p n n n n
  q_2: d d p p p n n n n n
  q_1 pairs: 2*(d, p) + 4*(d, n) + 8*(p, n) = 14
  q_2 pairs: 6*(d, p) + 10*(d, n) + 15*(p, n) = 31

RankNet: Learning interactions for ranking

- Uses a one-layer neural network and gradient descent
  - Inputs: Two query-document feature vectors
  - Output: Which document is ranked higher (binary)
- Advantage: Very standard neural model (Multilayer Perceptron in sklearn)
- Advantage: Allows interactions in features
- Disadvantage: Requires more training data
- Disadvantage: Not directly optimizing what we care about, e.g,. NDCG



RankNet and its descendants

- RankNet was backbone of Bing for several years
- LambdaRank designed to better directly optimize the model with search engine objective
  - Changes model weights with respect to the document's ideal position in the list
  - Highly ranked documents change the model's parameters more
- LambdaMART replaced the neural network with gradient boosted decision trees
  - You'll use this technique in your homework!
  - Covered in the discussion sections. Please attend them!

Pairwise approach Summary

- Advantages:
  - Handiness of applying existing classification methods
  - Ease of obtaining training instances of document pairs
    - E.g. click-through data from users
- Problems…
  - Learning objective is to minimize errors in classifying document pairs, not to minimize errors in ranking documents.
  - The number of document pairs varies largely from query to query, resulting biased models towards queries with more document pairs

List-wise Ranking

List-wise Ranking

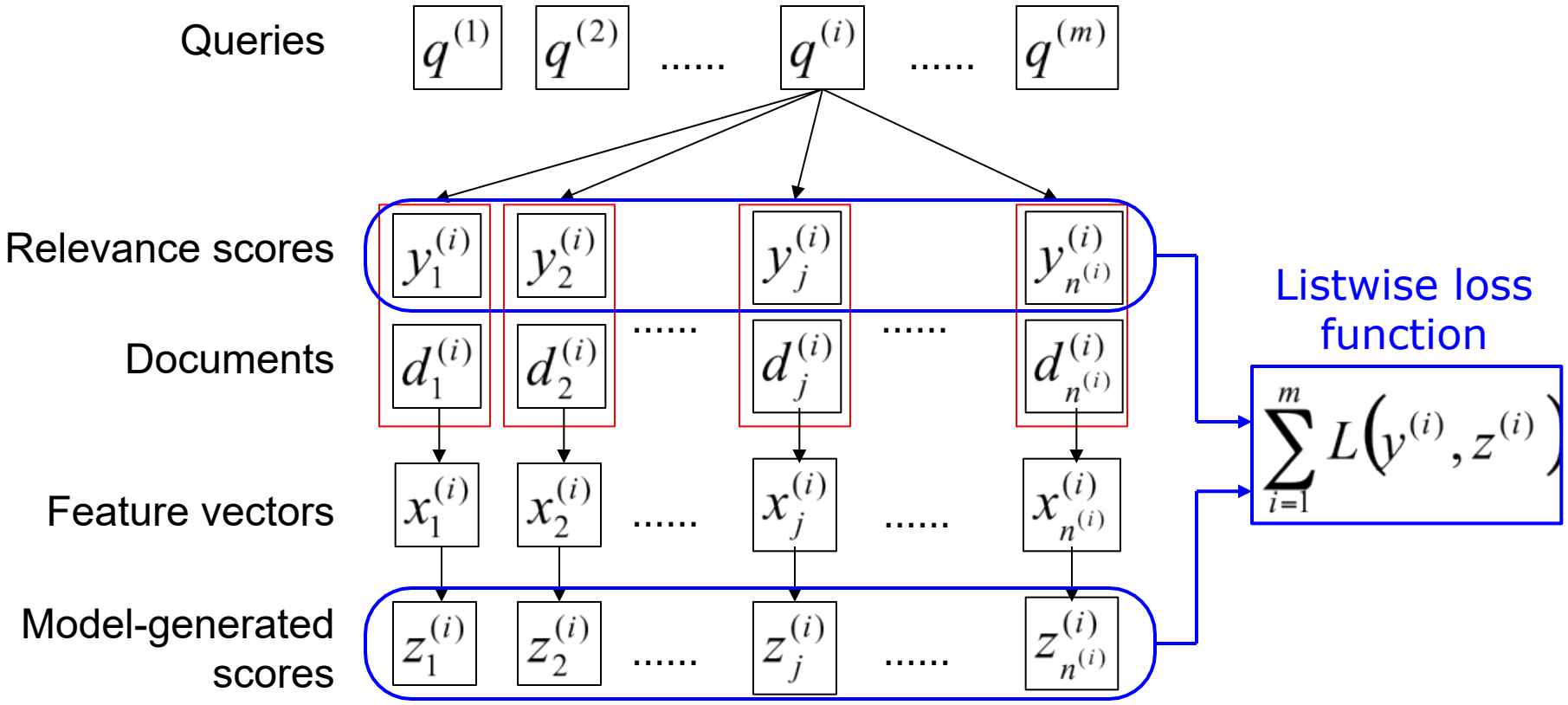
- Key idea: train the model to rank everything* at once
  - Often run only for the top-*k* documents
  - Side question: where do we get the "top-*k*" docs?
- Inputs: Lists of documents
- Outputs: Scores/ranks for each document
- Key question: How do we rank everything???
  - What does the "loss" function look like?

List-wise approach

- Training samples: document lists
- Listwise loss function
  - Represents the difference between the ranking list outputted by the ranking model and the ground truth ranking list
  - Probabilistic methods + cross-entropy
    - Permutation probability
    - Top k probability
- Classification model: neural network
- Optimization algorithm: gradient descent

List-wise approach

- List-wise framework:



Probability models for List-wise Ranking

- Map a list of relevance scores to a probability distribution. Two common approaches:
  - Permutation probability
  - Top *k* probability
- Q: Why do we need a probability distribution anyway?
  - A: A natural way to define loss function as we want gradient descent! Something differentiable
  - Commonly, this is cross-entropy

Permutation probability

- n objects are to be ranked
- A permutation π = ranking order of objects
- $\Omega_n = \{\pi_1, \pi_2, \ldots, \pi_n\}$ is a set of all possible permutations of n objects
- A list of scores $s = (s_1, s_2, \ldots, s_n)$

How likely is a permutation?

- Permutation probability is defined as:

$$P_s(\pi) = \prod_{j=1}^{n} \frac{\phi(s_{\pi(j)})}{\sum_{k=j}^{n} \phi(s_{\pi(k)})}$$

where

$\phi(.)$ = an increasing and strictly positive function

$s_{\pi(j)}$ = the score of object at position j of permutation π

- For example:

$$P_s(\langle 1,2,3,\dots,100\rangle) = \frac{\phi(s_1)}{\phi(s_1)+\phi(s_2)+\cdots+\phi(s_{100})} \cdot \frac{\phi(s_2)}{\phi(s_2)+\cdots+\phi(s_{100})} \cdot \dots \cdot \frac{\phi(s_{100})}{\phi(s_{100})}$$

Permutation probability (3/6)

- The permutation probability forms a probability distribution over
  - $P_s(\pi) > 0$ and $\sum_{\pi \in \Omega_n} P_s(\pi) = 1$
- The permutation with larger element in the front has higher probability
- If $s_1 > s_2 > s_3 > \cdots s_n$ (scores of each document) and <1,2,3,…,n> denotes a particular permutation
  - $p(< 1,2, \ldots, n >)$ has the highest probability
  - $p(< n, n-1, n-2, \ldots, 1 >)$ has the lowest probability

Example of Permutation Probabilities

- Example: 3 docs with relevance scores 3, 5, 10
  (higher is better)

$$P_s(\pi) = \prod_{j=1}^{n} \frac{\phi(s_{\pi(j)})}{\sum_{k=j}^{n} \phi(s_{\pi(k)})}$$

Permutation (π)	Probability (%)
3, 2, 1	34.72
2, 3, 1	21.37
3, 1, 2	20.83
1, 3, 2	11.11
2, 1, 3	6.41
1, 2, 3	5.56
Sum:	100.00

Problems with the permutation probability

- The number of permutation computation is of an order of $O(n!)$
  - The computation is intractable for large results sets!
- Consider the top k probability

Top k probability

- The probability of k objects (out of n objects) being ranked on the top k positions
- The top k subgroup $G_k(j_1, j_2, \ldots, j_k)$ is defined as a set containing all the permutations in which the top k objects are exactly $(j_1, j_2, \ldots, j_k)$
  - G_k is the collection of all the top k subgroups
  - G_k now has only $\frac{n!}{(n-k)!}$ elements << Ω_n
  - E.g. for 5 objects, the top 2 subgroup $G_2(1,3)$ includes:{(1,3,2,4,5), (1,3,2,5,4), (1,3,4,2,5), (1,3,4,5,2), (1,3,5,2,4), (1,3,5,4,2)}

Top *k* probability

- The top k probability of objects $(j_1, j_2, \ldots, j_k)$ is defined as:

$$P_s(G_k(j_1,j_2,\ldots,j_k)) = \sum_{\pi \in G_k(j_1,j_2,\ldots,j_k)} P_s(\pi)$$

- For example (5 objects):

$$\begin{aligned} P_s(G_2(1,3)) = {} & P_s(\langle 1,3,2,4,5 \rangle) + P_s(\langle 1,3,2,5,4 \rangle) \\ & + P_s(\langle 1,3,4,2,5 \rangle) + P_s(\langle 1,3,4,5,2 \rangle) \\ & + P_s(\langle 1,3,5,2,4 \rangle) + P_s(\langle 1,3,5,4,2 \rangle) \end{aligned}$$

- Do we still need to compute n! permutations?

Computing the top *k* probability

- The top k probability can be computed as follows:

$$P_s(G_k(j_1, j_2, \ldots, j_k)) = \prod_{t=1}^{k} \frac{\phi(s_{j_t})}{\sum_{l=t}^{n} \phi(s_{j_t})}$$

where

s_{j_t} = the score of object j_t (ranked at position t)

- For example (1,3,x,x,x):

$$P_s(G_2(s_1, s_3)) = \frac{\phi(s_1)}{\phi(s_1)+\phi(s_3)+\phi(s_2)+\phi(s_4)+\phi(s_5)} \cdot \frac{\phi(s_3)}{\phi(s_3)+\phi(s_2)+\phi(s_4)+\phi(s_5)}$$

List-wise loss function

- Cross-entropy between the top k distributions of two lists of scores:

$$L\left(y^{(i)}, z^{(i)}\right) = -\sum_{\forall g \in G_k} P_{y^{(i)}}(g) \log\left(P_{z^{(i)}}(g)\right)$$

where

i denotes the i^{th} query

$y^{(i)}$ denotes the ground truth list of scores

$z^{(i)}$ denotes the model-generated list of scores

- This is just pytorch's CrossEntropyLoss (if you know it)

Learning method: ListNet (1/2)

- A learning to rank method for optimizing the listwise loss function based on top ***k*** probability with neural network as the model and gradient descent as optimization algorithm
- f_ω denotes the ranking function based on the neural network model

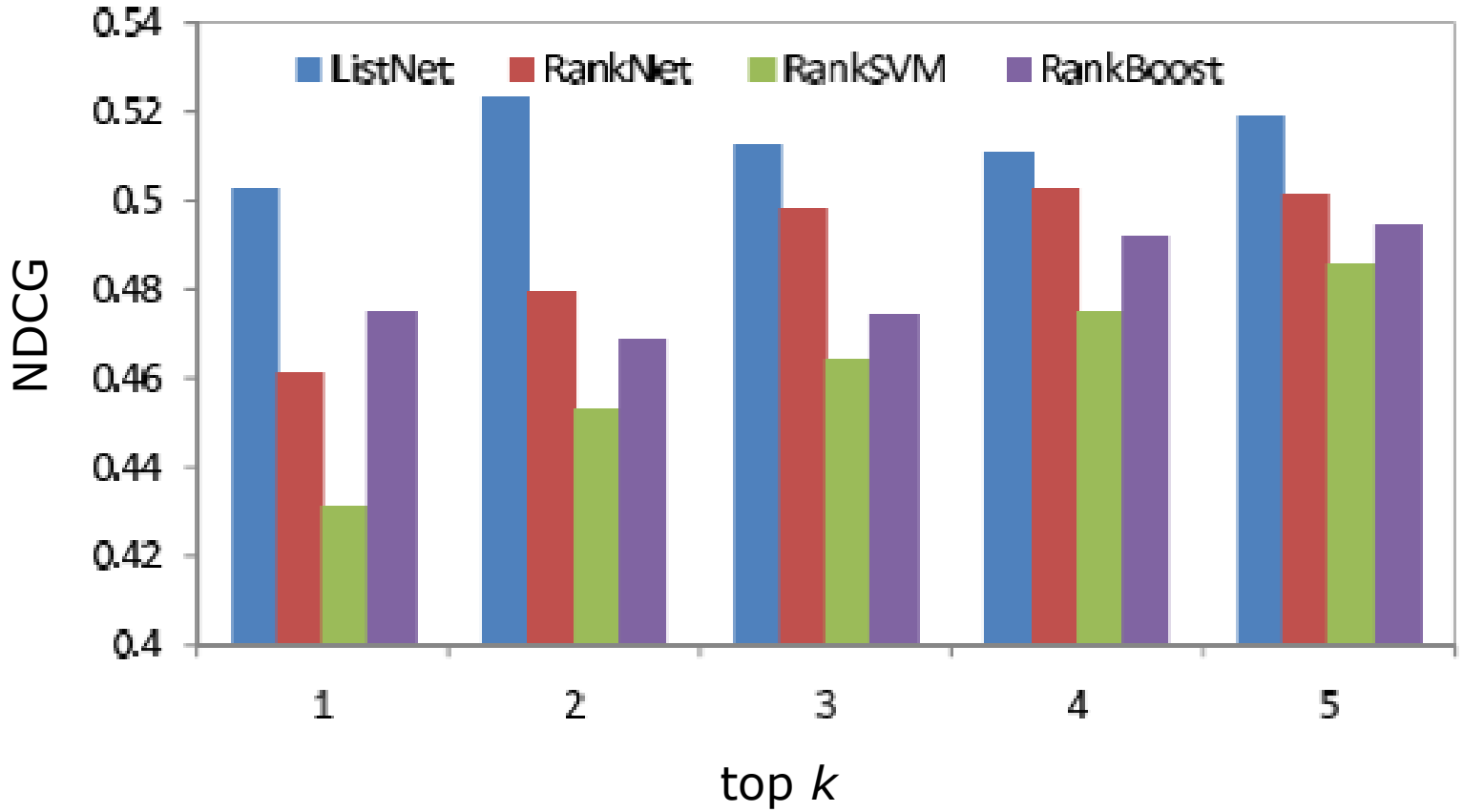
Learning method: ListNet

- Learning algorithm of ListNet:

Input: training data $\{(x_1,y_1), (x_2, y_2), \ldots (x_m, y_m)\}$
Parameter: number of iteration T and learning rate η
Initialize parameter ω
for $t = 1$ to T do
    for $i = 1$ to m do
        Input $x^{(i)}$ of query $q^{(i)}$ to neural network and compute score list $z^{(i)}(f_\omega)$ with current ω
        Compute gradient $\Delta\omega$ (w/ cross-entropy loss!)
        Update $\omega = \omega - \eta \times \Delta\omega$
    end for
end for
Output neural network model ω

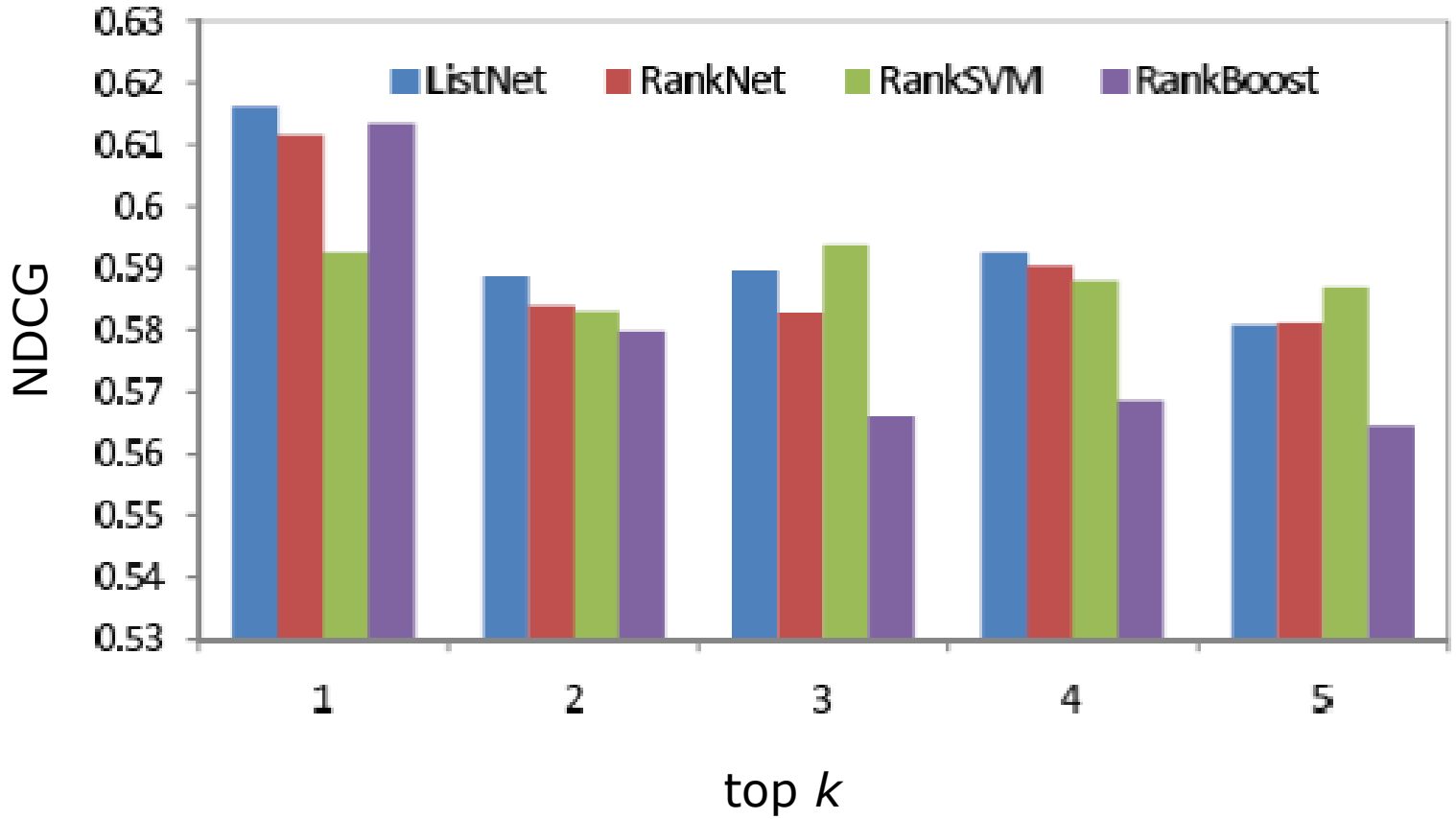
How good is ListNet vs Rank SVM?

- Ranking accuracies in terms of NDCG on TREC



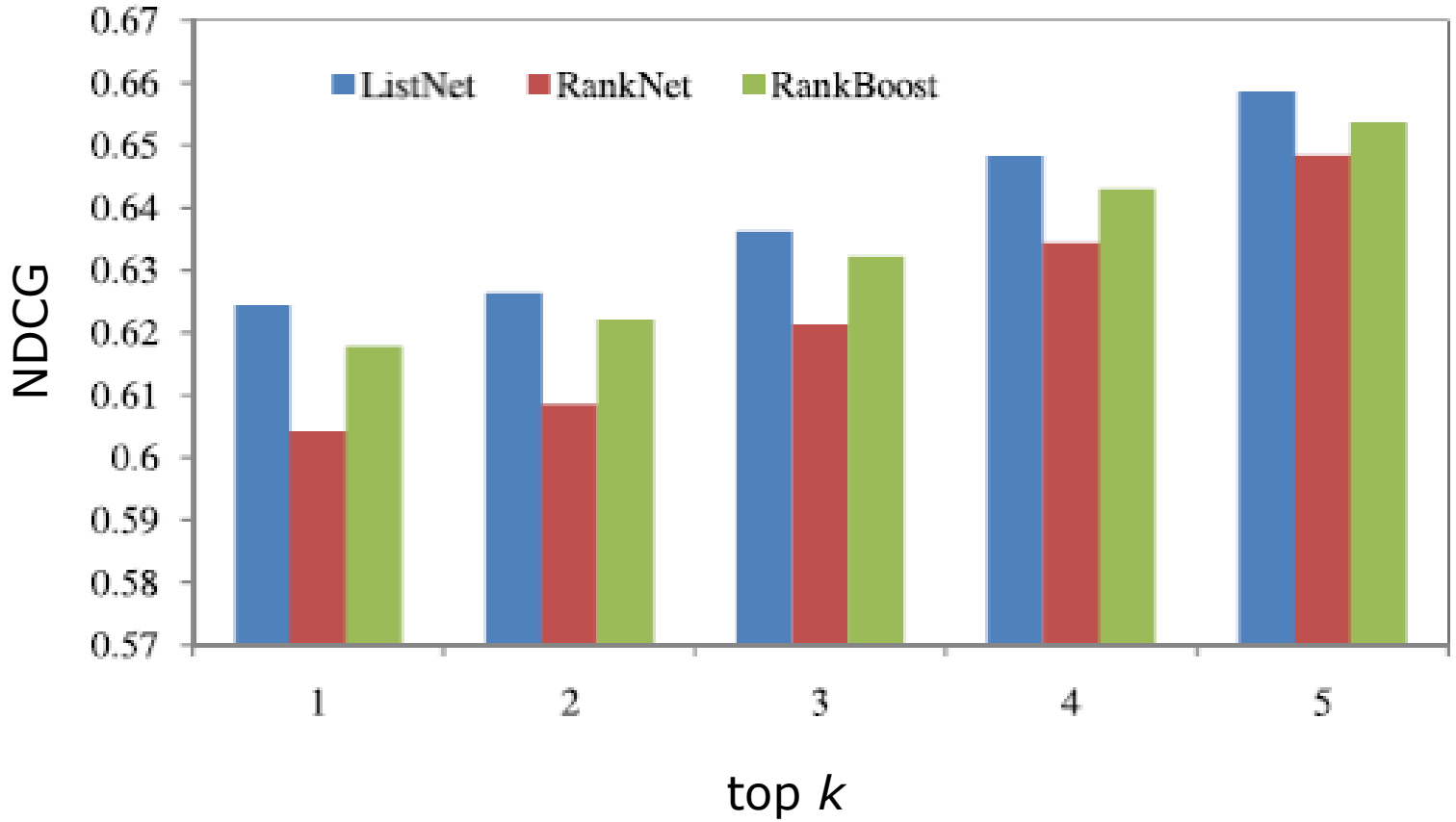
How good is ListNet vs Rank SVM?

- Ranking accuracies in terms of NDCG on OHSUMED



How good is ListNet vs Rank SVM?

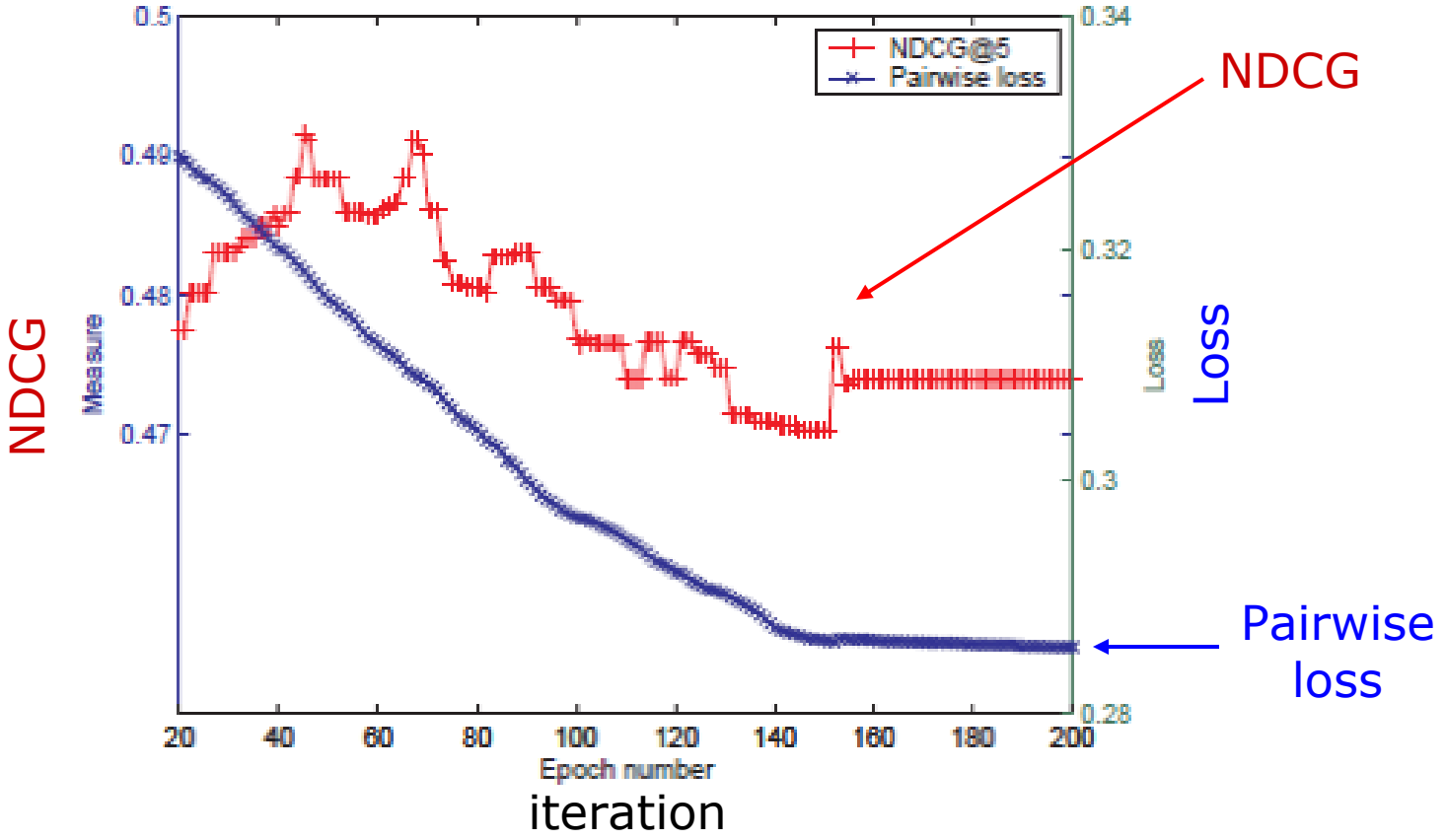
- Ranking accuracies in terms of NDCG on CSearch



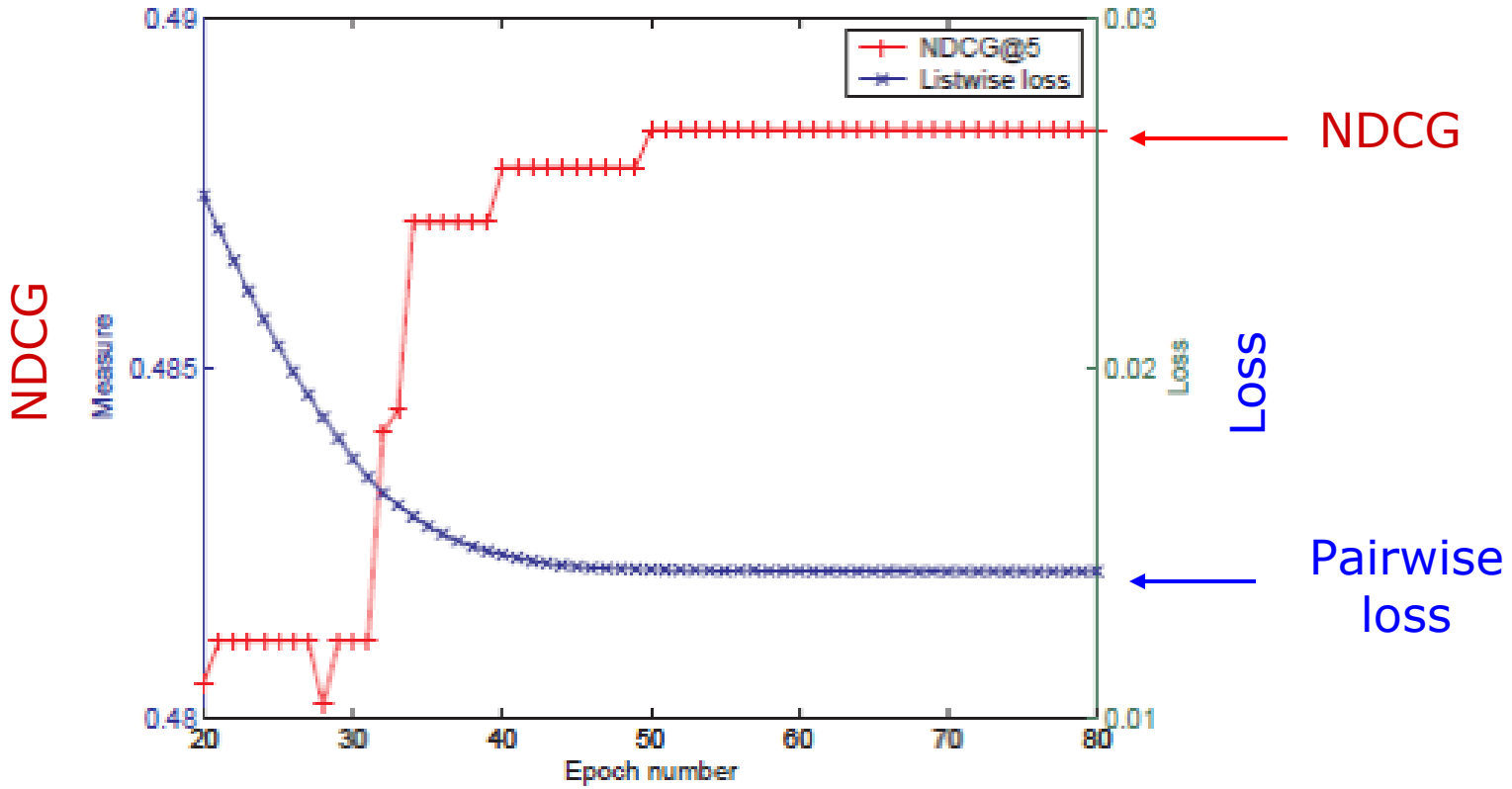
Discussion

- Pairwise approach employs a “pairwise” loss function, not suited for NCDG and MAP for performance measuring
- List-wise approach better represents the performance measures
- How do we know this? Observe the relationship between loss and NDCG in each iteration…

Pairwise loss vs. NDCG in RankNet



Listwise loss vs. NDCG in ListNet



You Should Know

- Gradient Descent is a way to update the parameters of a model
- A loss function is how we measure how far away a prediction is from the result we want
- Learning to Rank is a family of approaches
  - Pointwise predicts relevance (like a ranker!)
  - Pairwise compares items to predict which is ranked higher
  - Listwise ranks an entire ordering of the items
- Learning to Rank lets you incorporate a variety of features and use ML to figure out how to weight them