Get unlimited access Open in app

Published in Towards Data Science

Francesco Casalegno Follow
Feb 28 · 9 min read · Listen

Save

Learning to Rank: A Complete Guide to Ranking using Machine Learning

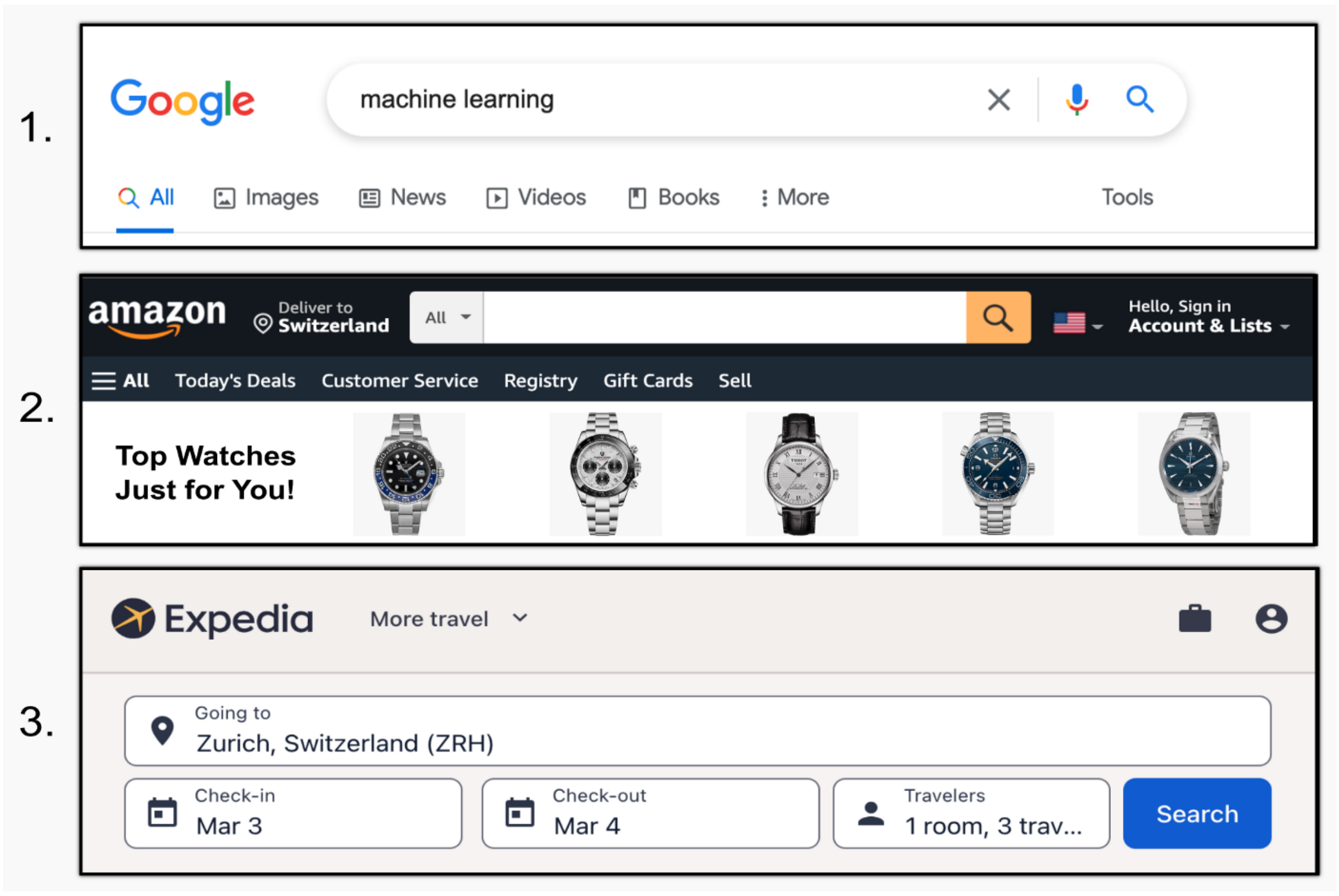
Photo by Nick Fewings on Unsplash

Ranking: What and Why?

In this post, by "**ranking**" we mean **sorting documents by relevance** to find contents of interest **with respect to a query**. This is a fundamental problem of **Information**

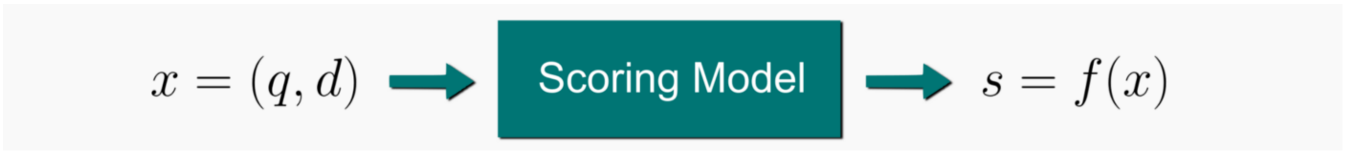
2. **Recommender Systems** — Given a user profile and purchase history, sort the other items to find new potentially interesting products for the user.

3. **Travel Agencies** — Given a user profile and filters (check-in/check-out dates, number and age of travelers, …), sort available rooms by relevance.



Ranking applications: 1) search engines; 2) recommender systems; 3) travel agencies. (Image by author)

Ranking models typically work by predicting a **relevance score** $s = f(x)$ for each input $x = (q, d)$ where **q is a query** and **d is a document**. Once we have the relevance of each document, we can sort (i.e. rank) the documents according to those scores.



Ranking models rely on a scoring function. (Image by author)

The scoring model can be implemented using various approaches.

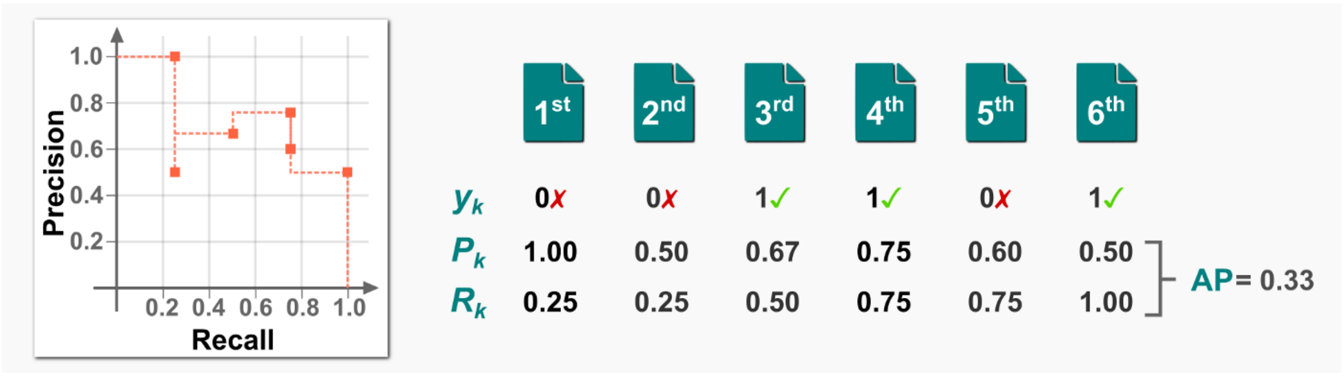
- **Learning to Rank** – The scoring model is a Machine Learning model that learns to predict a score ***s*** given an input ***x* = (*q*, *d*)** during a training phase where some sort of ranking loss is minimized.

In this article we focus on the latter approach, and we show how to implement **Machine Learning models for Learning to Rank**.

Ranking Evaluation Metrics

Before analyzing various ML models for Learning to Rank, we need to define which metrics are used to evaluate ranking models. These metrics are computed on the **predicted documents ranking**, i.e. the ***k*-th top retrieved document** is the *k*-th document with highest predicted score ***s***.

Mean Average Precision (MAP)



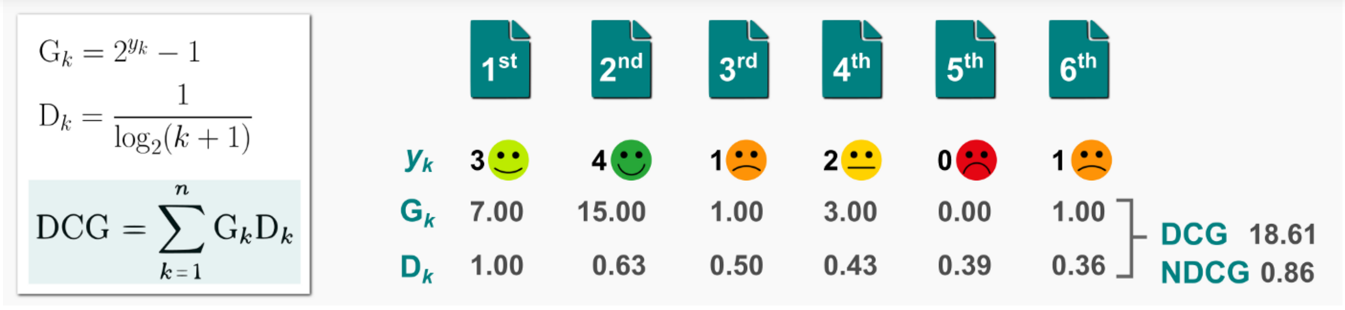
	1st	2nd	3rd	4th	5th	6th
y_k	0✗	0✗	1✓	1✓	0✗	1✓
P_k	1.00	0.50	0.67	0.75	0.60	0.50
R_k	0.25	0.25	0.50	0.75	0.75	1.00

MAP — Mean Average Precision. (Image by author)

Mean Average Precision is used for tasks with **binary relevance**, i.e. when the true score *y* of a document *d* can be only **0 (*non relevant*) or 1 (*relevant*)**.

For a given query *q* and corresponding documents $D = \{d_1, \ldots, d_n\}$, we check how many of the top *k* retrieved documents are relevant (y=1) or not (y=0)., in order to compute **precision** P_k and **recall** R_k. For $k = 1 \ldots n$ we get different P_k and R_k values that define the **precision-recall curve**: the area under this curve is the **Average Precision (AP)**.

Finally, by computing the average of AP values for a set of *m* queries, we obtain the **Mean Average Precision (MAP)**.



$$G_k = 2^{y_k} - 1$$

$$D_k = \frac{1}{\log_2(k+1)}$$

$$DCG = \sum_{k=1}^{n} G_k D_k$$

	1st	2nd	3rd	4th	5th	6th
y_k	3	4	1	2	0	1
G_k	7.00	15.00	1.00	3.00	0.00	1.00
D_k	1.00	0.63	0.50	0.43	0.39	0.36

DCG 18.61
NDCG 0.86

DCG — Discounted Cumulative Gain. (Image by author)

Discounted Cumulative Gain is used for tasks with **graded relevance**, i.e. when the true score y of a document d is a discrete value in a scale measuring the relevance w.r.t. a query q. A typical scale is **0 (*bad*), 1 (*fair*), 2 (*good*), 3 (*excellent*), 4 (*perfect*)**.

For a given query q and corresponding documents $D = \{d_1, \ldots, d_n\}$, we consider the the k-th top retrieved document. The **gain** $G_k = 2\hat{}y_k - 1$ measures how useful is this document (we want documents with high relevance!), while the **discount** $D_k = 1/\log(k+1)$ penalizes documents that are retrieved with a lower rank (we want relevant documents in the top ranks!).

The sum of the **discounted gain** terms $G_k D_k$ for $k = 1 \ldots n$ is the **Discounted Cumulative Gain (DCG)**. To make sure that this score is bound between 0 and 1, we can divide the measured DCG by the ideal score IDCG obtained if we ranked documents by the true value y_k. This gives us the **Normalized Discounted Cumulative Gain (NDCG)**, where NDCG = DCG/IDCG.

Finally, as for MAP, we usually compute the average of DCG or NDCG values for a set of m queries to obtain a mean value.

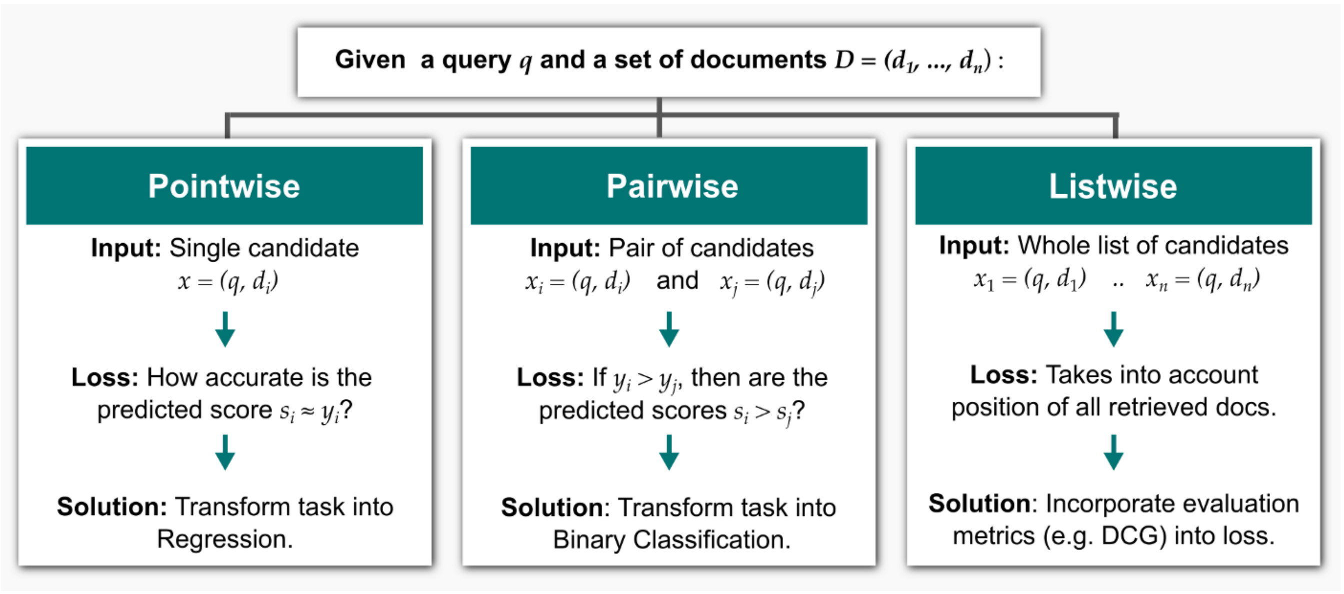
Machine Learning Models for Learning to Rank

To build a Machine Learning model for ranking, we need to define **inputs**, **outputs** and **loss function**.

- **Input** – For a query $\boldsymbol{q}$ we have $\boldsymbol{n}$ documents $\boldsymbol{D = \{d_1, \ldots, d_n\}}$ to be ranked by relevance. The elements $\boldsymbol{x_i = (q, d_i)}$ are the inputs to our model.
- **Output** – For a query-document input $x_i = (q, d_i)$, we assume there exists a true

element for Learning to Rank models. In general, we have **3 approaches**, depending on how the loss is computed.

1. **Pointwise Methods** – The total loss is computed as the sum of loss terms defined on **each document d_i** (hence ***pointwise***) as the distance between the predicted score s_i and the ground truth y_i, for $i=1 \ldots n$. By doing this, we transform our task into a **regression problem,** where we train a model to predict y.
2. **Pairwise Methods** – The total loss is computed as the sum of loss terms defined on **each pair of documents d_i, d_j** (hence ***pairwise***) , for $i, j=1 \ldots n$. The objective on which the model is trained is to predict whether $y_i > y_j$ or not, i.e. which of two documents is more relevant. By doing this, we transform our task into a **binary classification problem**.
3. **Listwise Methods** – The loss is directly computed on the whole list of documents (hence ***listwise***) with corresponding predicted ranks. In this way, ranking metrics can be more directly incorporated into the loss.


Given a query q and a set of documents $D = (d_1, \ldots, d_n)$:
Pointwise
Input: Single candidate $x = (q, d_i)$
Loss: How accurate is the predicted score $s_i \approx y_i$?
Solution: Transform task into Regression.
Pairwise
Input: Pair of candidates $x_i = (q, d_i)$ and $x_j = (q, d_j)$
Loss: If $y_i > y_j$, then are the predicted scores $s_i > s_j$?
Solution: Transform task into Binary Classification.
Listwise
Input: Whole list of candidates $x_1 = (q, d_1)$ .. $x_n = (q, d_n)$
Loss: Takes into account position of all retrieved docs.
Solution: Incorporate evaluation metrics (e.g. DCG) into loss.

Machine Learning approaches to Learning to Rank: pointwise, pairwise, listwise. (Image by author)

Pointwise Methods

The pointwise approach is the simplest to implement, and it was the first one to be proposed for Learning to Rank tasks. The loss directly measures the distance between ground true score y_i and predicted s_i so we treat this task by effectively solving a

$$L(\underline{s}, \underline{y}) = \sum_{i=1}^{n} (s_i - y_i)^2$$

MSE loss for pointwise methods as in Subset Ranking. (Image by author)

Pairwise Methods

The main issue with pointwise models is that true relevance scores are needed to train the model. But in many scenarios training data is available only with **partial information**, e.g. we only know which document in a list of documents was chosen by a user (and therefore is *more relevant*), but we don't know exactly *how relevant* is any of these documents!

For this reason, pairwise methods don't work with absolute relevance. Instead, they work with **relative preference**: given two documents, we want to predict if the first is more relevant than the second. This way we solve a **binary classification task** where we only need the ground truth y_{ij} ($=1$ if $y_i > y_j$, 0 otherwise) and we map from the model outputs to probabilities using a logistic function: $s_{ij} = \sigma(s_i - s_j)$. This approach was first used by **RankNet**, which used a Binary Cross Entropy (BCE) loss.

$$L(\underline{s}, \underline{y}) = -\sum_{i,j=1}^{n} y_{ij} \log(s_{ij}) + (1 - y_{ij}) \log(1 - s_{ij})$$

BCE loss for pairwise methods as in RankNet. (Image by author)

RankNet is an improvement over pointwise methods, but all documents are still given the same importance during training, while we would want to give **more importance to documents in higher ranks** (as the DCG metric does with the discount terms).

Unfortunately, rank information is available only after sorting, and sorting is non differentiable. However, to run Gradient Descent optimization we don't need a loss function, we only need its gradient! **LambdaRank** defines the gradients of an implicit loss function so that documents with high rank have much bigger gradients:

$$\lambda_i := \frac{\partial L}{\quad} = \frac{1}{\quad} \sum \sigma(s_i - s_j)|G_i - G_j||D_i - D_j|$$

Having gradients is also enough to build a Gradient Boosting model. This is the idea that **LambdaMART** used, yielding even better results than with LambdaRank.

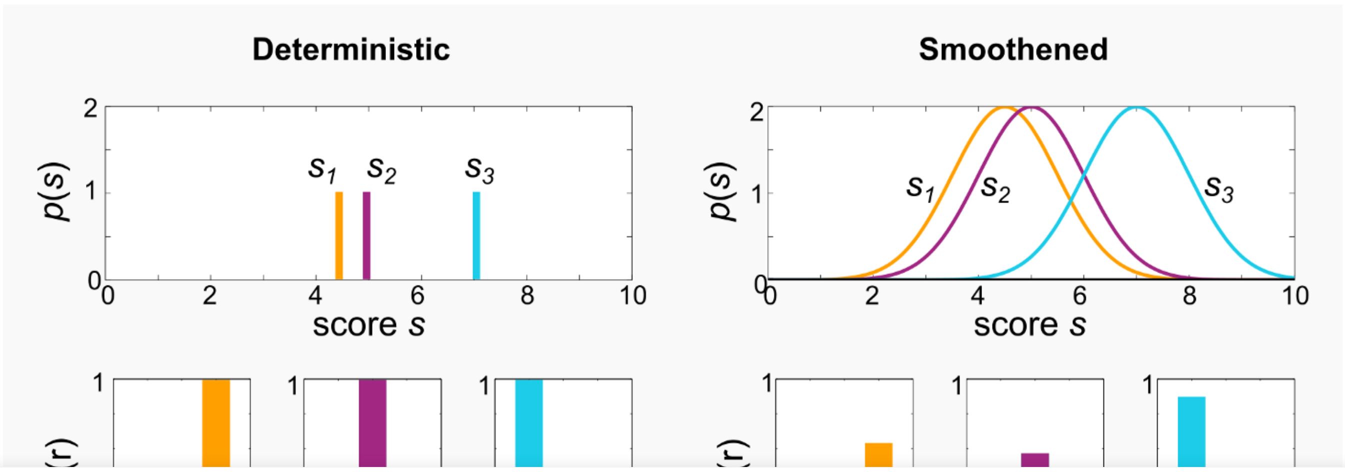
Listwise Methods

Pointwise and pairwise approaches transform the ranking problem into a surrogate regression or classification task. Listwise methods, in contrast, solve the problem more **directly by maximizing the evaluation metric**.

Intuitively, this approach **should give the best results**, as information about ranking is fully exploited and the NDCG is directly optimized. But the obvious problem with setting **Loss = 1 – NDCG** is that the rank information needed to compute the discounts D_k is only available after sorting documents based on predicted scores, and **sorting is non-differentiable**. How can we solve this?

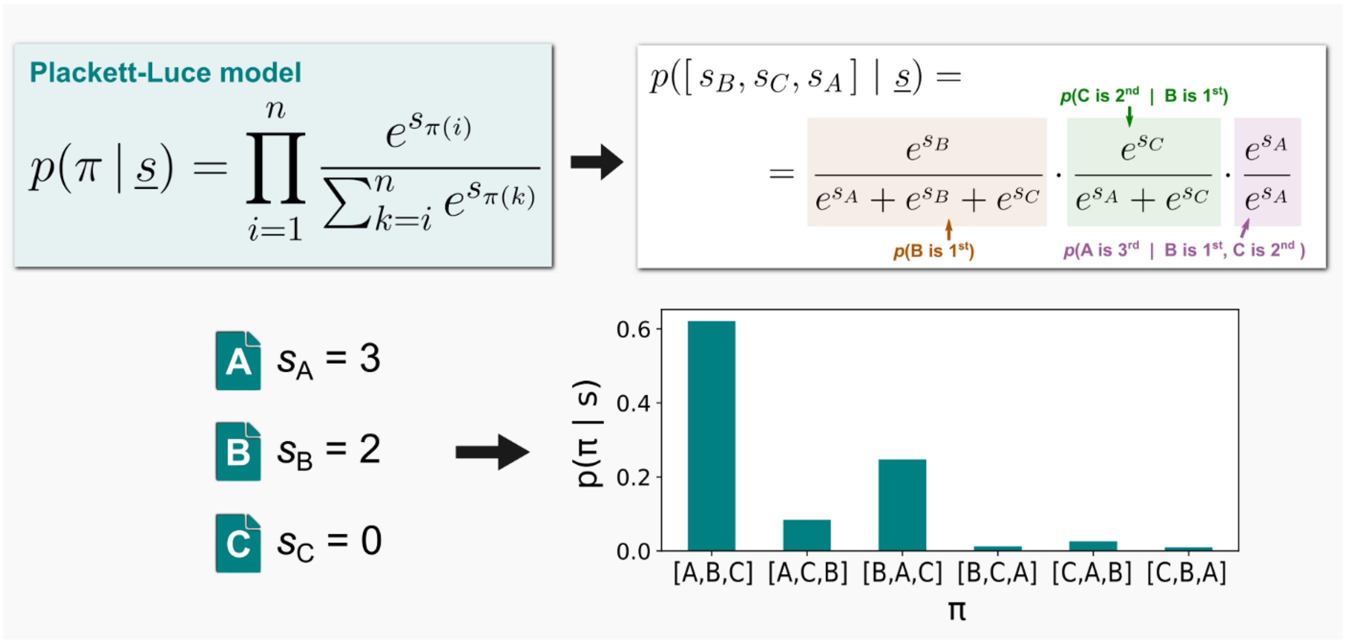
A **first approach** is to use an iterative method where **ranking metrics are used to re-weight** instances at each iteration. This is the approach used by **LambdaRank** and **LambdaMART**, which are indeed between the pairwise and the listwise approach.

A **second approach** is to approximate the objective to make it differentiable, which is the idea behind **SoftRank**. Instead of predicting a deterministic score $s = f(x)$, we predict a **smoothened probabilistic score** $s \sim \mathcal{N}(f(x), \sigma^2)$. The **ranks** $\boldsymbol{k}$ are non-continuous functions of **predicted scores** $\boldsymbol{s}$, but thanks to the smoothening we can compute **probability distributions for the ranks** of each document. Finally, we optimize **SoftNDCG**, the expected NDCG over this rank distribution, which is a smooth function.



Uncertainty in scores produce a smooth loss in SoftRank. (Image by author)

A **third approach** is consider that each ranked list corresponds to a permutation, and define a **loss over space of permutations**. In **ListNet**, given a list of scores *s* we define the probability of any permutation using the **Plackett-Luce model**. Then, our loss is easily computed as the **Binary Cross-Entropy distance** between true and predicted **probability distributions over the space of permutations.**


Plackett-Luce model
$p(\pi \mid \underline{s}) = \prod_{i=1}^{n} \frac{e^{s_{\pi(i)}}}{\sum_{k=i}^{n} e^{s_{\pi(k)}}}$
$p([s_B, s_C, s_A] \mid \underline{s}) = \frac{e^{s_B}}{e^{s_A}+e^{s_B}+e^{s_C}} \cdot \frac{e^{s_C}}{e^{s_A}+e^{s_C}} \cdot \frac{e^{s_A}}{e^{s_A}}$
p(C is 2nd | B is 1st)
p(B is 1st)
p(A is 3rd | B is 1st, C is 2nd)
A $s_A = 3$
B $s_B = 2$
C $s_C = 0$
p(π | s)
0.6
0.4
0.2
0.0
[A,B,C] [A,C,B] [B,A,C] [B,C,A] [C,A,B] [C,B,A]
π

Probability of various permutation using Plackett-Luce model in ListNet. (Image by author)

Finally, the **LambdaLoss** paper introduced a new perspective on this problem, and created a **generalized framework** to define new listwise loss functions and achieve **state-of-the-art accuracy**. The main idea is to frame the problem in a rigorous and general way, as a **mixture model** where the ranked list π is treated as a hidden variable. Then, the loss is defined as the negative log likelihood of such model.

$$L(\underline{s}, \underline{y}) = -\log p(\underline{y} \mid \underline{s}) = -\log \sum_{\pi \in \Pi} \underbrace{p(\underline{y} \mid \underline{s}, \pi)}_{\text{likelihood}} \cdot \underbrace{p(\pi \mid \underline{s})}_{\text{ranked list distribution}}$$

LambdaLoss loss function. (Image by author)

The authors of the LambdaLoss framework proved two essential results.

obtained by accurately choosing the **likelihood** $p(y \mid s, \pi)$ and the **ranked list distribution** $p(\pi \mid s)$.

2. This framework allows us to define metric-driven loss functions directly connected to the ranking metrics that we want to optimize. This allows to **significantly improve the state-of-the-art on Learningt to Rank tasks.**

Conclusions

Ranking problem are found everywhere, from information retrieval to recommender systems and travel booking. Evaluation metrics like MAP and NDCG take into account both rank and relevance of retrieved documents, and therefore are difficult to optimize directly.

Learning to Rank methods use Machine Learning models to predicting the relevance score of a document, and are divided into 3 classes: pointwise, pairwise, listwise. On most ranking problems, listwise methods like LambdaRank and the generalized framework LambdaLoss achieve state-of-the-art.

References

- Wikipedia page on "Learning to Rank"
- Li, Hang. "A short introduction to learning to rank." 2011
- L. Tie-Yan. "Learning to Rank for Information Retrieval", 2009
- L. Tie-Yan "Learning to Rank", 2009
- X. Wang, "The LambdaLoss Framework for Ranking Metric Optimization", 2018
- Z. Cao, "Learning to rank: from pairwise approach to listwise approach", 2007
- M Taylor, "SoftRank: optimizing non-smooth rank metrics", 2008