Introduction to Learning to Rank

2019-06-07

Kyle Chung

In this session, we introduce learning to rank (LTR), a machine learning sub-field applicable to a variety of real world problems that are related to ranking prediction or candidate recommendation. We will walk through the evolution of LTR research in the past two decades, illustrate the very basic concept behind the theory. In the end we will also live-demo how we can quickly build a proof-of-concept LTR model using real world data and two of the state-of-the-art machine learning open source libraries.

Outline

- Introduction to Learning-to-Rank (LTR)
  - What is LTR and what's the difference between it and other ML models?
  - The classical problem (And also the non-classical ones)
  - Different types of LTR modeling approach
- How to Evaluate a Ranking Model?
- The Evolution of mainstream LTR
  - RankNet -> LambdaNet -> LambdaMART -> LambdaLoss
- Demo with the go-to open source libraries
  - LambdaMART with `lightgbm` (Gradient Boosting Trees)
  - Listwise LTR with `tensorflow` (Deep Neural Nets)

What is Learning to Rank (LTR)?

> Learning to rank refers to machine learning techniques for training a model to solve a ranking task. Usually it is a supervised task and sometimes semi-supervised.

Regression vs Classification vs LTR

They are all supervised learning. But the target variables differ. The learning algorithm will differ in how we mathematically formulate the learning objective.

Regression

We try to learn a function $f(x)$ given feature x to predict a real-valued $y \in \mathbb{R}$.

Example: *I want to predict the average CASA balance in the next 7 days for an account.*

Classification

We try to learn a function $f(x)$ given feature x to predict a set of discrete integer label $y \in \{1, 2, \dots, N\}$ with N-classes.

Example: *I want to predict if a given FD (Fixed Deposit) will be renewed at its current expire.*

Learning to Rank

We try to learn a function $f(q, D)$, given a query q and a relevant list of items D, to predict the order (ranking) of all items within list.

A Classical Problem in LTR

Web Search Ranking

Given a search query, rank the relevance of the resulting matched document URLs, such that more relevant document should be presented first to the user.

More formally, we depict the above problem as the following task:

Given a query q, and the resulting n documents $D = d_1, d_2, \dots, d_n$, we'd like to learn a function f such that $f(q, D)$ will predict the relevance of any given document associated with a query. Ideally, $f(q, D)$ should return an ordered list of documents D^*, ranked from the most to least relevant to the given query q.

Popular web search datasets for large-scale LTR benchmark:

- Microsoft LTR Datasets (https://www.microsoft.com/en-us/research/project/mslr/?from=http%3A%2F%2Fresearch.microsoft.com%2Fen-us%2Fprojects%2Fmslr%2F)
- Yahoo LTR Datasets (https://webscope.sandbox.yahoo.com/catalog.php?datatype=c)

Non-Classical Problems in LTR

LTR is a general approach for solving ranking task. Here are some examples other than just web search ranking. Note that not all of them are obviously a ranking task in the first glance.

- **Recommender system** (Solve personal product perference ranking)
- **Stock portfolio selection** (Solving equity return ranking)
- **Message auto reply** (Solving best-candidate ranking in email/message reply recommendation)
- **Image to text** (Solving best-candidate contextual feature)

General Types of LTR algorithm:

- Pointwise
- Pairwise
- Listwise

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

The Ranking Task

Given a query q, and the resulting n document $D = d_1, d_2, \dots, d_n$, we'd like to learn a function f such that $f(q, D)$ will predict the relevance of any given document associated with a query.

Pointwise LTR

In pointwise approach, the above ranking task is re-formulated as a regression (or classification) task. The function to be learned $f(q, D)$ is simplied as $f(q, d_i)$. That is, the relevance of each document given a query is scored independently. In recent literature $f(q, d_i)$ is called pointwise scoring function, while $f(q, D)$ is refered to as groupwise scoring function.

If we have two queries associated with 2 and 3 resulting matching documents, respectively:

$$\begin{aligned} q_1 &\rightarrow d_1, d_2 \\ q_2 &\rightarrow d_3, d_4, d_5 \end{aligned}$$

Then the training examples x_i in a pointwise framework will decouple them into every query-document pair:

$$\begin{aligned} x_1 &: q_1, d_1 \\ x_2 &: q_1, d_2 \\ x_3 &: q_2, d_3 \\ x_4 &: q_2, d_4 \\ x_5 &: q_2, d_5 \end{aligned}$$

Since each document is indeed scored independently with the absolute relevance as the target label (could be real-valued in order or simply binary), **the task is entirely no difference than a traditional regression or classification task.** Any such machine learning algorithm can be applied to pointwise solution.

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

The Ranking Task

Given a query q, and the resulting n document $D = d_1, d_2, \ldots, d_n$, we'd like to learn a function f such that $f(q, D)$ will predict the relevance of any given document associated with a query.

Pairwise LTR

In pairwise approach, we are still trying to learn the *pointwise* scoring function $f(q, d_i)$, however, our training examples are now consructed by pairs of documents within the same query:

$x_1 : q_1, (d_1, d_2)$
$x_2 : q_2, (d_3, d_4)$
$x_3 : q_2, (d_3, d_5)$
$x_4 : q_2, (d_4, d_5)$

Given such setup, a new set of pairwise BINARY labels can be derived, by simply comapring the individual relevance score in each pair. For example, given the first query q_1, if $y_1 = 0$ (totally irrelevant) for d_1 and $y_2 = 3$ (highly relevant) for d_2, then we have a new label $y_1 < y_2$ for the document pair (d_1, d_2). **Now the problem has become a binary classification learning task.**

In order to learn the still-pointwise function $f(q, d_i)$ in a pairwise manner, we model the score difference probablistically:

$$Pr(i \succ j) \equiv \frac{1}{1 + exp^{-(s_i - s_j)}}$$

In plain words, if document i is better matched than document j (which we denote as $i \succ j$), then the probability of the scoring function to have scored $f(q, d_i) = s_i$ higher than $f(q, d_j) = s_j$ should be close to 1. Put it differnetly, the model is trying to learn, given a query, how to score a pair of document such that a more relevant document should be scored higher.

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

The Ranking Task

Given a query q, and the resulting n document $D = d_1, d_2, \dots, d_n$, we'd like to learn a function f such that $f(q, D)$ will predict the relevance of any given document associated with a query.

Listwise LTR

The first ever proposed listwise approach is ListNet. Here we explain how it approach the ranking task.

ListNet is based on the concept of permutation probability given a ranking list. Again we assume there is a *pointwise* scoring function $f(q, d_i)$ used to score and hence rank a given list of items. But instead of modeling the probability of a pairwise comparison using scoring difference, now we'd like to model the probability of the entire ranking results.

$x_1 \ : q_1, (d_1, d_2)$
$x_2 \ : q_2, (d_3, d_4, d_5)$

Permutation Probability

Let's denote π as a particular permutation of a given list of length n, $\phi(s_i) = f(q, d_i)$ as any increasing function of scoring s_i given query q and document i. The probability of having a permutation π can be written as:

$$Pr(\pi) = \prod_{i=1}^{n} \frac{\phi(s_i)}{\sum_{k=i}^{n} \phi(s_k)}$$

To illustrate, given a list of 3 items, the probability of returning the permutation s_1, s_2, s_3 is calculated as: $Pr(\pi = \{s_1, s_2, s_3\}) = \frac{\phi(s1)}{\phi(s1)+\phi(s2)+\phi(s3)} \cdot \frac{\phi(s2)}{\phi(s2)+\phi(s3)} \cdot \frac{\phi(s3)}{\phi(s3)}$.

Top-One Probability

Due to computational complexity, ListNet simplies the problem by looking at only the top-one probability of a given item. The top-one probability of object i equals the sum of the permutation probabilities of permutations in which object i is ranked on the top. Indeed, the top-one probability of object i can be written as:

$$Pr(i) = \frac{\phi(s_i)}{\sum_{k=1}^{n} \phi(s_k)}$$

Now given any two ranking list represented by top-one probabilities, we are able to measure their difference using cross entropy. Then we can build a machine learning algorithm that minimize that cross entropy.

For the choice of function $\phi(\cdot)$, it can be as simple as just an exponential function. Indeed, when $\phi(\cdot)$ is expotential and list length is two, the solution will reduce to a pairwise model we just depicted in the previos section.

- Pros:
  - Theoretically sound solution to approach a ranking task.
- Cons:
  - Costly to compute in its theoretical form and hence several approximations are used in practice. (For example the use of top-one probability.)
  - Scoring function is still pointwise, which could be sub-optimal.

Evaluation of LTR Model

How to we evaluate a result of ranking prediction?

Several metrics have been proposed and commonly used in the evaluation of a ranking model:

- Binary Relevance
  - Mean Average Precision (MAP)
  - Mean Reciprocal Rank (MRR)
- Graded Relevance
  - Normalized Discounted Cumulative Gain (NDCG)
  - Expected Reciprocal Rank (ERR)

In general, binary measures only consider relevant v.s. irrelevant, while graded measures will also consider the ranking among relevant items. The degree of relevancy matters in this case when scoring a ranking list.

Mean Average Precision, MAP

MAP is a measure based on binary label of relevancy. The calculation of MAP is indeed NOT that straightforward. First we need to define *precision at k given a query* $P@k(q)$ as:

$$P@k(q) \equiv \frac{\sum_{i=1}^{k} r_i}{k}$$

for an ordered list of prediction r_i for all k items. $r_i = 1$ if it is relevant and 0 otherwise.

Then we define the *average precision given a query* $AP(q)$ at k items as:

$$AP(q)@k \equiv \frac{1}{\sum_{i=1}^{k} r_i} \sum_{i=1}^{k} P@i(q) \times r_i$$

Mean Average Precision is just the mean of $AP(q)$ for all queries:

$$MAP \equiv \frac{\sum_{q=1}^{Q} AP(q)}{Q}$$

Note that MAP is order-sensitive due to the introduction of the term r_i in the calculation of AP. Intuitively, it is doing the average of precision at each ranking position, but penalizing the precision at positions with irrelevant item by strcitly setting them to zeroes.

Example:

$$q_1 \rightarrow d_1, d_2$$
$$q_2 \rightarrow d_3, d_4, d_5$$

Assuming only d_2, d_3, d_5 are relevant document given their corresponding query.

AP of query 1: $\frac{1}{1} \times (\frac{0}{1} \times 0 + \frac{1}{2} \times 1) = \frac{1}{2}$

AP of query 2: $\frac{1}{2} \times (\frac{1}{1} \times 1 + \frac{1}{2} \times 0 + \frac{2}{3} \times 1) = \frac{5}{6}$

MAP: $\frac{1}{2} \times (\frac{1}{2} + \frac{5}{6}) \approx 67\%$

Caveat:

MAP is order-sensitive, but only in a binary context: relevant items should come first than irrelevant ones. It does not take into account the optimal ranking among only relevant items. In the above example, even if d_5 is prefered than d_3 (and both are relevant), average precision of the query is the same for d_3, d_4, d_5 and d_5, d_4, d_3. But ideally the latter should be scored higher.

Reciproal Rank, RR

RR focuses on the first correctly predicted relevant item in a list. Given a ranking list, assume r_i is the rank of the highest ranking relevant item. Say, if the the 2nd item is the first relevant item in the list, RR is $\frac{1}{2}$ for this query.

Mean Reciproal Rank, MRR

By definition, each query will have a reciprocal rank. MRR is simply the average of RR for all queries:

$$MRR \equiv \frac{1}{Q} \sum_{i=1}^{Q} \frac{1}{r_i}$$

Expected Reciproal Rank, ERR

The underlying rationale is the empirical finding in the web search problem that *the likelihood a user examines the document at rank i is dependent on how satisfied the user was with previously observed documents in the list*. ERR hence tries to quantify the usefulness of a document at rank i conditioned on the degree of relevance of documents at rank less than i.

Assume the probability of a user finding the result is satisfied at position i in a given list is denoted as R_i. The likelihood of a session for which the user is satisfied and stops at position r is: $\prod_{i=1}^{r-1}(1 - R_i)R_r$.

Now we model R_i such that it is an increasing function of relevance:

$$R = R(g) \equiv \frac{2^g - 1}{2^{g_{max}}}$$

where g is the labeled (graded) relevance such that $g \in \{0, 1, \ldots, g_{max}\}$. $g = 0$ suggests irrelevant and $g = g_{max}$ perfectly relevant.

Finally, ERR is defined as:

$$ERR \equiv \sum_{r=1}^{n} \frac{1}{r} R_r \prod_{i=1}^{r-1}(1 - R_i)$$

Here $\frac{1}{r}$ can be considered as a utility function $\tau(r)$ that satisfies $\tau(1) = 1$ and $\tau(r) \rightarrow 0$ as $r \rightarrow \infty$.

Note that ERR is a measure on a list with a single query, so the corresponding de-generated measure is RR instead of MRR. To evaluate on results from multiple queries, we will need to further average ERRs among queries.

Example:

$$\begin{aligned} &q_1 \rightarrow d_1, d_2 \\ &q_2 \rightarrow d_3, d_4, d_5 \end{aligned}$$

Assuming only d_2, d_3, d_5 are relevant document given their corresponding query.

MRR: $(\frac{1}{2} + \frac{1}{1}) \times \frac{1}{2} = \frac{3}{4}$

ERR of $q_1: 0 + \frac{1}{2} \times \frac{2^1-1}{2^1} \times (1 - \frac{2^0-1}{2^1}) = \frac{1}{4}$

ERR of q_2: $\frac{1}{1} \times \frac{2^1-1}{2^1} + 0 + \frac{1}{3} \times \frac{2^1-1}{2^1} \times (1 - \frac{2^0-1}{2^1}) \times (1 - \frac{2^1-1}{2^1}) = \frac{7}{12}$

Caveat:

MRR is a binary measure, while ERR is a graded measure. Also MRR and ERR is NOT directly comparable between each other. Both being graded measure, ERR is less popular than NDCG in empirical works due to

Normalized Discounted Cumulative Gain, NDCG

First we define Discounted Cumulative Gain at position k as:

$$DCG@k \equiv \sum_{i=1}^{k} \frac{2^{l_i} - 1}{log_2(i+1)}$$

where l_i is the grading of relevance at rank i.

To intuitively understand the metric, the numerator is simply an increasing function of relevance, the more relevant the higher. This is the *gain* from each item. The denominator is a decreasing function of ranking position, this is the *discounted* component of the metric. Put together, higher relevance gains more points, but the lower it is ranked the higher also the discount. In the end the metric will prefer higher relevant item to be ranked higher, which is exactly the desired ranking property we'd like to pursue.

Normalized DCG is then defined as:

$$NDCG@k = \frac{DCG@k}{IDCG@k}$$

where $IDCG@k$ is the *Ideal* DCG@k given the result. It is the DCG@k calculated by re-sorting the given list by its true relevance labels. That is, IDCG@k is the maximum possible DCG@K value one can get given a ranking list.

Example:

$$q_1 \rightarrow d_1, d_2$$
$$q_2 \rightarrow d_3, d_4, d_5$$

Assuming only d_2, d_3, d_5 are relevant document given their corresponding query.

NDCG of q_1: $\frac{0+\frac{2^1-1}{log_2 3}}{\frac{2^1-1}{log_2 2}+0} = \frac{1}{log_2 3} \approx 0.631$

NDCG of q_2: $\frac{\frac{2^1-1}{log_2 2}+0+\frac{2^1-1}{log_2 4}}{\frac{2^1-1}{log_2 2}+\frac{2^1-1}{log_2 3}+0} = \frac{1.5}{1+\frac{1}{log_2 3}} \approx 0.92$

Labeling Issues

Broadly speaking there are two approaches to label a ranking dataset:

- Human judgement
- Derivation from user behavior log

For the 1st approach, massive manpower is required to label the relevance of each item given a query. In real world lots of dataset cannnot be labeled in such way, so we rely on the 2nd approach which indirectly infer user preference among different items.

Usually pairwise preference can be infered from user interaction with the query result. For example, use click data to infer web search relevance. This is also why pairwise approach in LTR can gain much more attention than the pointwise method: due to data availability.

Evolution of LTR

From Pointwise to Pairwise

In the literature of LTR, a set of very important theoretical and also empirical works were done by Chris Burges (http://chrisburges.net/) from MicroSoft Research, who have established the very foundation of the pairwise approach in LTR.

Those important pairwise LTR models include:

- RankNet (2005)
- LambdaNet (2006)
- LambdaMART (2007); high quality implementation available in the library `lightgbm` (https://github.com/Microsoft/LightGBM)

From Pairwise to Listwise, and More

Recently, researchers from Google generalize the LambdaMART framework to provide a theoretical background of the ranking model of all 3 types of loss function (pointwise, pairwise, listwise) and the direct optimization of all the popular ranking metrics (NDCG, MAP, ...). The framework is called LambdaLoss (2018).

A production-ready implementation of such framework is also open-sourced as a ranking module (https://github.com/tensorflow/ranking) under the popular library `TensorFlow`. A *groupwise* scoring function is also proposed and can be implemented in the library.

The reason why we choose specifically to elaborate the above mentioned models is because they are the very foundation of LTR literature, cited more than a thousand times.

And the reason why the libraries are chosen is basically the same: they are the state-of-the-art popular open source go-to frameworks in this field.

A Digression: What is Machine Learning?

ML is nothing more than *mathematical optimization*. But in plain words, what is ML?

Here is a metaphor of ML:

- There is a function $f(\cdot)$, or an estimator, also called a **learner**
- The job of a learner is to *learn*, and then to *guess* the correct anwser to a given question
- Learner is characterized by model parameters W; value of W shapes learner's opinion about the answer
- Here is how learner guess:
  1. Setup a value for W
  2. Give out a guess based on that W
  3. Calculate how far the guess is from the truth, this is measured by **loss**
- When loss is high (meaning the guess is not even close), learner repeatedly tries the above steps until loss cannot be further reduced
- Learner stops at W^* which can give out the best guess

ML is a science of how we should efficiently perform the above exercise.

Real-life analogy? **Number Guessing**

I'm an machine learning expert.

Interviewer: What's your biggest strength?

Me: I'm an machine learning expert.

Interviewer: What's 9 + 10?

Me: It's 3.

Interviewer: Not even close. It's 19.

Me: It's 16.

Interviewer: It's still 19.

Me: It's 18.

Interviewer: No. It's 19.

Me: It's 19.

Interviewer: *You're hired.*

RankNet

Remember that we model the score difference between a given pair (i, j) as a probability based on the sigmoid function:

$$Pr(i \succ j) = P_{ij} \equiv \frac{1}{1 + exp^{-(s_i - s_j)}}$$

where

$$s_i = f(q, d_i)$$

is the pointwise score output by our underlying learner $f(q, d)$, which in RankNet is formulated as a *2-layer neural network* parameterized by a set of w_k. (Or even think simplier, let $f(q, d_i) = wx_i$ as a linear learner.)

Given a probability distribution p, the entropy is defined as: $p \cdot log_2 \frac{1}{p}$. Now let $y_{ij} \in \{0, 1\}$ be the actual label of the given pair (i, j), The loss function of the above setup will be the *cross entropy*:

$$loss = -\sum_{i \neq j} y_{ij} log_2 P_{ij} + (1 - y_{ij}) log_2 (1 - P_{ij})$$

The cross entropy measures how close two probability distribution are to each other. So naturally it is a good objective function for a machine learning model that models probability to optimize. Using backprop techinque we can numerically find the model weights in $f(q, d)$ that minimize the cross entropy loss.

Note that the above loss is very general: it is just the expected log-loss, or the sum of cross entropy from each training record, used to measure how good the model distribution is approximating the empirical distribution of the traing data (which in turn serves as an approximation to the unknown true distribution generating the training data). We can easily swap the neural network with other learners, resulting in a variety of different pairwise LTR models.

LambdaNet

Two important enhancements have been achieved from RankNet to LambdaNet.

1. Training speed-up thanks to factorization of gradient calculation
2. Optimization towards a ranking metric

Gradient Factorization

For the first point, LambdaNet is a **mathematically improved version of RankNet.** The improvement is based on a factorization of the calculation of gradient of the cross entropy loss, under its pairwise update context.

Given the point cross entropy loss as L:

$$L = y_{ij} log_2 P_{ij} + (1 - y_{ij}) log_2 (1 - P_{ij})$$

The gradient (the 1st-order derivative of the loss w.r.t. a model parameter w_k) can be written as:

$$\frac{\partial L}{\partial w_k} = \frac{\partial L}{\partial s_i}\frac{\partial s_i}{\partial w_k} + \frac{\partial L}{\partial s_j}\frac{\partial s_j}{\partial w_k}$$

In plain words, the impact of a change in model parameter w_k will go through the resulting changes in the model scores and then the changes in loss. Now rewrite the gradient in total losses for all training pairs $\{i, j\}$ that satisfied $i \succ j$:

$$\begin{aligned}\frac{\partial L_T}{\partial w_k} &= \sum_{\{i,j\}}\left[\frac{\partial L}{\partial s_i}\frac{\partial s_i}{\partial w_k} + \frac{\partial L}{\partial s_j}\frac{\partial s_j}{\partial w_k}\right] \\ &= \sum_i \frac{\partial s_i}{\partial w_k}\left(\sum_{\forall j \prec i}\frac{\partial L(s_i, s_j)}{\partial s_i}\right) + \sum_j \frac{\partial s_j}{\partial w_k}\left(\sum_{\forall i \succ j}\frac{\partial L(s_i, s_j)}{\partial s_j}\right)\end{aligned}$$

with the fact that:

$$\frac{\partial L(s_i, s_j)}{\partial s_i} = -\frac{\partial L(s_i, s_j)}{\partial s_j} = log_2 e\left[(1 - y_{ij}) - \frac{1}{1 + e^{s_i - s_j}}\right],$$

and a re-indexing of the second-term, we end up with:

$$\begin{aligned}\frac{\partial L_T}{\partial w_k} &= \sum_i \frac{\partial s_i}{\partial w_k}\left[\sum_{\forall j \prec i}\frac{\partial L(s_i, s_j)}{\partial s_i} + \sum_{\forall j \prec i}\frac{\partial L(s_j, s_i)}{\partial s_i}\right] \\ &= \sum_i \frac{\partial s_i}{\partial w_k}\left[\sum_{\forall j \prec i}\frac{\partial L(s_i, s_j)}{\partial s_i} - \sum_{\forall j \succ i}\frac{\partial L(s_j, s_i)}{\partial s_j}\right] \\ &= \sum_i \frac{\partial s_i}{\partial w_k}\lambda_i\end{aligned}$$

The intuition behind the above gradient:

> For each document in a given query, there is a gradient component we denoted as lambda, which is calculated by considering all the superior and inferior documents comparing to it. A relatively worse document will push the current document up, and a relatively better one will push it down.

Ranking Metric Optimization

Since we model the score difference of a pair of documents in a query as a probability measure, the model is optimizing the pairwise correctness of ranking, which may not be the ultimately desirable objective.

Remember that the ranking objective is indeed measured by (ideally) a position-sensitive graded measure such as NDCG. But in the above setup NDCG is not directly linked to the minimization of cross entropy. A straightforward and also simple solution is to use NDCG as an early stop criteria and determine by using a validation dataset.

LambdaRank proposes yet another solution. The researcher found that during the gradient update using the lambda notion, for each pair instead of calculating just the lambda, we can adjusted lambda by the change in NDCG for that pair provided that the position of the two item swaped with each other.

The lambda of a given document is:

$$\begin{aligned}\lambda_i &= \Bigg[\sum_{\forall j \prec i} \frac{\partial L(s_i, s_j)}{\partial s_i} - \sum_{\forall j \succ i} \frac{\partial L(s_j, s_i)}{\partial s_j}\Bigg] \\ &= \Bigg[\sum_{\forall j \prec i} \lambda_{ij} - \sum_{\forall j \succ i} \lambda_{ij}\Bigg]\end{aligned}$$

The proposed method is to adjust the pairwise lambda λ_{ij} such that:

$$\lambda_{ij} \equiv \frac{\partial L(s_i, s_j)}{\partial s_i} \cdot |\Delta NDCG_{ij}|$$

where $\Delta NDCG_{ij}$ is the change in NDCG when the position of i and j are swapped.

The researcher found that by such adjustment, without theoretical proof, the model is empirically optimizing NDCG, and hence yield better overall results.

LambdaMART

LambdaMART is simply a LambdaNet but replaces the underlying neural network model with **gradient boosting regression trees** (or more general, gradient boosting machines, GBM). GBM is proven to be very robust and performant in handling real world problem.

The model wins several real-world large-scale LTR contests.

LambdaLoss

In the original LambdaRank and LambdaMART framework, no theoretical work has been done to mathematically prove that ranking metric is being optimized after the adjustment of the lambda calculation. The finding is purely based on empirical works, i.e., by observing the results from varying dataset and simulation with experiments.

Researchers from Google recently (2018) published a generalized framework called LambdaLoss, which serves as an extension of the original ranking model and comes with a thorough theoretical groundwork to justify that the model is indeed optimizing a ranking metric.

Live Demo

Using Python to quickly experiment with LTR models.

Demo Dataset: Yahoo LTR Dataset (https://webscope.sandbox.yahoo.com/catalog.php?datatype=c)

In [1]:

```
# Example of one training instance in the Yahoo LTR dataset.
!head -n1 data/ltrc_yahoo/set1.train.txt
```

```
0 qid:1 10:0.89028 11:0.75088 12:0.01343 17:0.4484 18:0.90834 21:0.7
7818 27:0.71505 29:0.77307 30:0.75925 39:0.65244 43:0.79394 44:0.884
06 45:0.87946 66:0.63696 69:0.24961 70:0.41479 71:0.82664 74:0.78587
77:0.70475 83:0.55661 85:0.7212 86:0.93081 91:0.34635 98:0.70029 10
1:0.96287 108:0.19608 122:0.85512 123:0.18873 124:0.46599 127:0.4343
4 129:0.048097 133:0.44747 139:0.92407 145:0.46578 146:0.57682 147:
0.63382 149:0.84404 154:0.73041 155:0.50316 159:0.30671 170:0.89536
172:0.66755 173:0.40438 174:0.78512 177:0.88064 178:0.54927 179:0.51
912 187:0.2056 192:0.47747 195:0.59496 197:0.83402 204:0.80444 208:
0.50368 212:0.81822 216:0.29351 222:0.48201 227:2.6864e-05 235:0.216
3 239:0.88083 241:0.21466 242:0.39518 243:0.68992 245:0.76751 246:1.
592e-05 247:0.76309 253:0.54512 265:0.70421 266:0.26977 271:0.34358
276:0.58251 281:0.91145 282:0.75332 287:1.592e-05 300:0.43013 302:0.
66278 305:0.22842 320:0.48644 325:0.54927 326:0.81728 329:0.63642 33
2:0.25652 339:0.86446 341:0.66553 344:0.067189 347:0.4434 348:0.4042
7 349:0.83413 350:0.69668 353:0.83216 356:0.26036 359:0.19518 374:2.
1889e-05 376:0.3086 377:3.2834e-05 382:0.58666 383:0.54927 384:0.228
42 385:0.22501 387:0.54927 395:0.51588 397:0.78664 399:0.2066 405:0.
86375 417:0.90713 418:0.087714 426:0.021835 427:0.51798 431:0.88408
433:0.76635 436:0.80754 437:0.83927 438:0.82536 439:0.84688 441:0.81
958 442:0.31634 445:0.45793 446:0.30368 450:0.35208 451:0.20139 454:
0.84541 465:0.609 468:3.6814e-05 473:0.92185 474:0.36567 476:0.43744
480:0.30985 481:0.7371 483:0.79078 486:0.59763 487:0.92985 488:3.581
9e-05 499:0.23455 504:0.62898 513:0.81697 514:0.42767 515:0.44476 51
8:0.74652 527:0.77818 533:0.27975 535:0.43961 538:0.74678 540:0.8611
541:0.46581 558:0.72917 570:0.91611 571:0.89485 574:0.6785 575:0.704
14 578:0.26276 579:0.31934 585:0.74199 586:0.4434 595:0.228 596:0.47
662 603:0.73649 606:0.61591 607:0.85539 610:0.85791 616:0.64438 627:
0.54927 636:1.6915e-05 637:0.38146 638:0.2845 640:0.68023 641:0.5999
3 642:0.35761 648:0.98273 654:0.89037 657:0.79396 658:0.9292 665:0.7
8185 669:0.074045 671:0.69325 674:0.61727 677:0.94189 690:0.21568 69
1:0.89071 692:0.77798 693:0.80534 694:0.76245 697:0.046567 699:0.516
53
```

Implement LambdaMART using `lightgbm`

We will use `lightgbm` (https://github.com/microsoft/LightGBM) to demonstrate how we can build a PoC LTR model within 50 lines of Python code.

In [2]:

```
import lightgbm as lgb

# Note that we have convert the original raw data into a pure libsvm format.
# For more details, pls refer to: https://github.com/guolinke/boosting_tree_benc
hmarks/tree/master/data
infile_train = "data/ltrc_yahoo/yahoo.train"
infile_valid = "data/ltrc_yahoo/yahoo.test"

train_data = lgb.Dataset(infile_train)
valid_data = lgb.Dataset(infile_valid)

# Set group info.
# We can igonre the step if *.query files exist with input files in the same di
r.
train_group_size = [l.strip("\n") for l in open(infile_train + ".query")]
valid_group_size = [l.strip("\n") for l in open(infile_valid + ".query")]
train_data.set_group(train_group_size)
valid_data.set_group(valid_group_size)

# Parameters are borrowed from the official experiment doc:
# https://lightgbm.readthedocs.io/en/latest/Experiments.html
param = {
    "task": "train",
    "num_leaves": 255,
    "min_data_in_leaf": 1,
    "min_sum_hessian_in_leaf": 100,
    "objective": "lambdarank",
    "metric": "ndcg",
    "ndcg_eval_at": [1, 3, 5, 10],
    "learning_rate": .1,
    "num_threads": 2
}

res = {}
bst = lgb.train(
    param, train_data,
    valid_sets=[valid_data], valid_names=["valid"],
    num_boost_round=50, evals_result=res, verbose_eval=10)
```

```
[10]	valid's ndcg@1: 0.693756	valid's ndcg@3: 0.693717
valid's ndcg@5: 0.714933	valid's ndcg@10: 0.762077
[20]	valid's ndcg@1: 0.698652	valid's ndcg@3: 0.701907
valid's ndcg@5: 0.722211	valid's ndcg@10: 0.76766
[30]	valid's ndcg@1: 0.704456	valid's ndcg@3: 0.707324
valid's ndcg@5: 0.727423	valid's ndcg@10: 0.771926
[40]	valid's ndcg@1: 0.708184	valid's ndcg@3: 0.711492
valid's ndcg@5: 0.731144	valid's ndcg@10: 0.775175
[50]	valid's ndcg@1: 0.712476	valid's ndcg@3: 0.715018
valid's ndcg@5: 0.733456	valid's ndcg@10: 0.77714
```

In [3]:

```
# Show the eval metric in tabular format.

import pandas as pd
pd.DataFrame(res["valid"]).tail()
```

Out[3]:

	ndcg@1	ndcg@10	ndcg@3	ndcg@5
45	0.711949	0.776528	0.714129	0.732383
46	0.711911	0.776486	0.713903	0.732672
47	0.711707	0.776844	0.714586	0.732831
48	0.711209	0.776886	0.714442	0.732866
49	0.712476	0.777140	0.715018	0.733456

Implement Listwise LTR using `tensorflow`

We will use `tensorflow` (https://github.com/tensorflow/tensorflow) along with `tensorflow_ranking` (https://github.com/tensorflow/ranking) to demonstrate how we can build a PoC LTR model within 200 lines of Python code.

Note that usually using `tensorflow` involves much more effort since it is a lower-level framework for machine learning modeling.

In [4]:

```
import tensorflow as tf
import tensorflow_ranking as tfr

import warnings
warnings.filterwarnings("ignore")
tf.logging.set_verbosity(tf.logging.ERROR)

# The code here largely borrows from:
# https://github.com/tensorflow/ranking/blob/master/tensorflow_ranking/examples/
tf_ranking_libsvm.ipynb
# Comparing to lightgbm, we need much more effort to build a model using tf.
# This is because tf is in general a lower-level framework for machine learning
 modeling,
# particularly deep neural nets.

tf.enable_eager_execution()
assert tf.executing_eagerly()

_NUM_FEATURES = 699
_LIST_SIZE = 99
_BATCH_SIZE = 32
_HIDDEN_LAYER_DIMS=["20", "10"]
_LOSS = tfr.losses.RankingLossKey.APPROX_NDCG_LOSS


# Input reader.
def input_fn(path):
    train_dataset = tf.data.Dataset.from_generator(
        tfr.data.libsvm_generator(path, _NUM_FEATURES, _LIST_SIZE, seed=777),
        output_types = (
            {str(k): tf.float32 for k in range(1, _NUM_FEATURES + 1)},
            tf.float32
        ),
    output_shapes = (
        {str(k): tf.TensorShape([_LIST_SIZE, 1]) for k in range(1, _NUM_FEATURES
+ 1)},
        tf.TensorShape([_LIST_SIZE])
    ))
    train_dataset = train_dataset.shuffle(1000).repeat().batch(_BATCH_SIZE)
    return train_dataset.make_one_shot_iterator().get_next()


# Test the reader.
infile_train_svm = "data/ltrc_yahoo/set1.train.txt"
infile_valid_svm = "data/ltrc_yahoo/set1.test.txt"

d = input_fn(infile_train_svm)
print(d[1])  # A label matrix of batch size X list size.
```

```
tf.Tensor(
[[ 0.  1.  0. ... -1. -1. -1.]
 [ 0.  0.  1. ... -1. -1. -1.]
 [ 2.  2.  2. ... -1. -1. -1.]
 ...
 [ 2.  2.  1. ... -1. -1. -1.]
 [ 1.  2.  2. ... -1. -1. -1.]
 [ 0.  0.  1. ... -1. -1. -1.]], shape=(32, 99), dtype=float32)
```

In [5]:

```
# Let's print the first feature tensor.
print(d[0]["1"])
```

```
tf.Tensor(
[[[0.     ]
  [0.69466]
  [0.     ]
  ...
  [0.     ]
  [0.     ]
  [0.     ]]

 [[0.64695]
  [0.67991]
  [0.67991]
  ...
  [0.     ]
  [0.     ]
  [0.     ]]

 [[0.65579]
  [0.66026]
  [0.67597]
  ...
  [0.     ]
  [0.     ]
  [0.     ]]

 ...

 [[0.74142]
  [0.74142]
  [0.74142]
  ...
  [0.     ]
  [0.     ]
  [0.     ]]

 [[0.6877 ]
  [0.67377]
  [0.     ]
  ...
  [0.     ]
  [0.     ]
  [0.     ]]

 [[0.64424]
  [0.58896]
  [0.6146 ]
  ...
  [0.     ]
  [0.     ]
  [0.     ]]], shape=(32, 99, 1), dtype=float32)
```

In [6]:

```
# Define feature column, which serves as an api for adding features into estimat
ors.
def example_feature_columns():
    feature_names = [
        "%d" % (i + 1) for i in range(0, _NUM_FEATURES)
    ]
    return {
        name: tf.feature_column.numeric_column(
            name, shape=(1,), default_value=0.0) for name in feature_names
    }


# Define the network structure.
# It is simply a fully-connected multi-layer perceptron.
def make_score_fn():
    """Returns a scoring function to build `EstimatorSpec`."""

    def _score_fn(context_features, group_features, mode, params, config):
        """Defines the network to score a documents."""
        del params
        del config
        # Define input layer.
        example_input = [
            tf.layers.flatten(group_features[name])
            for name in sorted(example_feature_columns())
        ]
        input_layer = tf.concat(example_input, 1)

        cur_layer = input_layer
        for i, layer_width in enumerate(int(d) for d in _HIDDEN_LAYER_DIMS):
            cur_layer = tf.layers.dense(
                cur_layer,
                units=layer_width,
                activation="tanh")
        logits = tf.layers.dense(cur_layer, units=1)
        return logits

    return _score_fn


def eval_metric_fns():
    """Returns a dict from name to metric functions."""
    metric_fns = {}
    metric_fns.update({
        "metric/ndcg@%d" % topn: tfr.metrics.make_ranking_metric_fn(
            tfr.metrics.RankingMetricKey.NDCG, topn=topn)
        for topn in [1, 3, 5, 10]
    })

    return metric_fns


def get_estimator(hparams):
    """Create a ranking estimator.

    Args:
      hparams: (tf.contrib.training.HParams) a hyperparameters object.

    Returns:
      tf.learn `Estimator`.
    """
    def _train_op_fn(loss):
        """Defines train op used in ranking head."""
        return tf.contrib.layers.optimize_loss(
            loss=loss,
            global_step=tf.train.get_global_step(),
            learning_rate=hparams.learning_rate,
            optimizer="Adagrad")

    ranking_head = tfr.head.create_ranking_head(
        loss_fn=tfr.losses.make_loss_fn(_LOSS),
        eval_metric_fns=eval_metric_fns(),
        train_op_fn=_train_op_fn)

    return tf.estimator.Estimator(
        model_fn=tfr.model.make_groupwise_ranking_fn(
            group_score_fn=make_score_fn(),
            group_size=1,
            transform_fn=None,
            ranking_head=ranking_head),
        params=hparams)
```

In [7]:

```
# Initialize the model.
hparams = tf.contrib.training.HParams(learning_rate=0.1)
ranker = get_estimator(hparams)

# Train.
ranker.train(input_fn=lambda: input_fn(infile_train_svm), steps=300)

# Evaluate.
ranker.evaluate(input_fn=lambda: input_fn(infile_valid_svm), steps=100)

# Monitor the tensorboard:
# tensorboard --logdir=<ranker.model_dir output>
```

Out[7]:

```
{'labels_mean': 1.231954,
 'logits_mean': -1.814632,
 'loss': -0.8202545,
 'metric/ndcg@1': 0.6314836,
 'metric/ndcg@10': 0.71695346,
 'metric/ndcg@3': 0.6387912,
 'metric/ndcg@5': 0.6630989,
 'global_step': 300}
```

References

Ranking Metrics

- Chapelle, Olivier, et al. "Expected reciprocal rank for graded relevance." Proceedings of the 18th ACM conference on Information and knowledge management. ACM, 2009.

LTR Models

- Ai, Qingyao, et al. "Learning groupwise scoring functions using deep neural networks." arXiv preprint arXiv:1811.04415 (2018).
- Burges, Christopher, et al. "Learning to rank using gradient descent." Proceedings of the 22nd International Conference on Machine learning (ICML-05). 2005.
- Burges, Christopher J., Robert Ragno, and Quoc V. Le. "Learning to rank with nonsmooth cost functions." Advances in neural information processing systems. 2007.
- Burges, Christopher JC. "From ranknet to lambdarank to lambdamart: An overview." Learning 11.23-581 (2010): 81.
- Cao, Zhe, et al. "Learning to rank: from pairwise approach to listwise approach." Proceedings of the 24th international conference on Machine learning. ACM, 2007.
- Li, Hang. "A short introduction to learning to rank." IEICE TRANSACTIONS on Information and Systems 94.10 (2011): 1854-1862.
- Wang, Xuanhui, et al. "The lambdaloss framework for ranking metric optimization." Proceedings of the 27th ACM International Conference on Information and Knowledge Management. ACM, 2018.