Learning to (Retrieve and) Rank — Intuitive Overview — part III

Michele Trevisiol

Real case candidate/job opening *ranking framework* at Jobandtalent (JT)

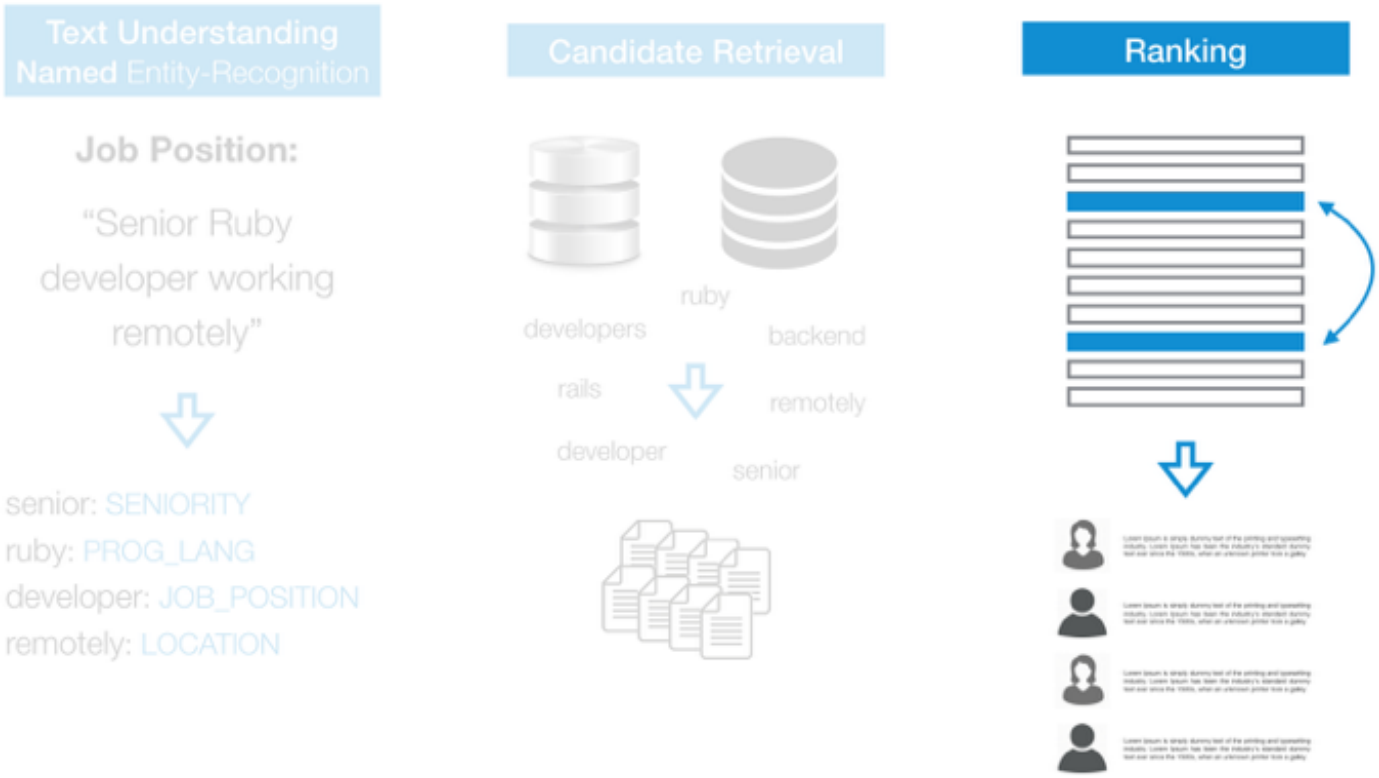

Fig1.

In previous posts (part I and II) we *have seen* how to build a candidate matrix `D`, filled with a set of *online* and *offline* features, given a job position `qi ∈ Q`. This is a collection of a set of information regarding the *matching* between that job and its candidates, and the candidate and company's *historical behaviors*. What we need now is an algorithm that can interpret this data and, based on the ***candidate relevance***, optimize the order of the returned candidates.

One thing that we didn't mention is the need for collecting the **relevance score** for each `<job, candidate>` pair. The relevance very much depends on what we want to optimise, at **Jobandtalent** we are taking care of the entire funnel, from

the *search* till the *offer signed* (directly within the app), thus we can define our relevance scores based on how far a candidate went within the funnel, *e.g.*, $y1$: shortlisted, $y2$: contacted, ..., yk: interviewed, ..., yn: hiring succeed. Where each score $yj \in \mathbb{N}$ (e.g., $\{y1=1, y2=2, \ldots, yn=n\}$).

This class of problems is known as **Ranking Problem**, and the most popular set of *supervised Machine Learning* methods that aim to solve them is called "***Learning to Rank***" (LTR).

Learning to Rank

There are *three main approaches* when dealing with the Ranking Problem, called ***Pointwise***, ***Pairwise*** and ***Listwise***, that we briefly summarise below.

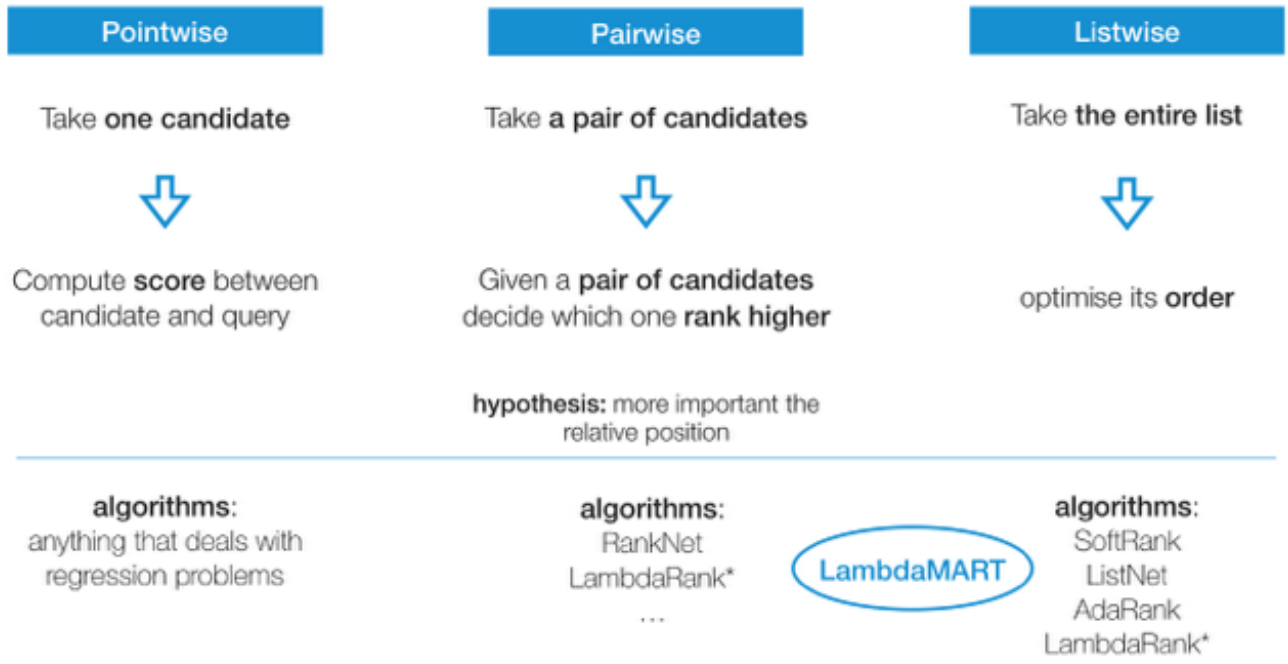

Fig2. Summary of the three main approaches of Learning to Rank.

For our *use-case,* we decided to use **LambdaMART** (TechReport, Microsoft 2010), the last of three popular algorithms (RankNet ICML2005, LambdaRank NIPS2006) main authored by *Chris Burges.*

Briefly, **RankNet** introduces the use of the Gradient Descent (GD) to learn the

learning function (update the *weights* or *model parameters*) for a **Learning to Rank** problem. Since the GD requires the calculation of gradient, RankNet requires a model for which the output is a *differentiable function* — meaning that its derivative always exists at each point in its domain (they use *neural networks* but it can be any other model with this property). RankNet is a ***pairwise approach*** and uses the GD to update the model parameters in order to minimise the cost (RankNet was presented with the Cross-Entropy cost function). This is like defining the ***force*** and the ***direction*** to apply when updating the positions of the two candidates (the one ranked higher *up* in the list while the other one *down* but with *the same force*). As an optimisation final decision, they speed up the whole process using the *Mini-batch Stochastic Gradient Descent* (computing all the weight updates for a given query, before actually applying them).

The black arrows denote the RankNet gradients with the size of the change, while the red arrows identify just the gradients–the directions of the changes (image taken from the original paper).

Note that when using RankNet, two cost functions are usually applied: one for optimization (*e.g.*, Cross-Entropy) and one for the final ranking quality (nDCG, MRR, MAP, *etc.*). This is quite common in classification and regression problems since the former cost function needs to respect more strict constraints in order to be easily optimized (smooth, convex, *etc.*), but it is the latter one the most interesting for the task in which the model is finally applied (this is defined by Burges as the *target cost*).

LambdaRank is based on the idea that we can use the same ***direction*** (gradient estimated from the candidates pair, defined as *lambda*) for the swapping, but scaling it by the *change* of the ***final metric***, such as *nDCG,* at each step (*e.g.*, swapping the pair and immediately computing the nDCG delta). This is a very tractable approach since it supports any model (with differentiable output) with the ranking metric we want to optimize in our use case.

LambdaMART is inspired by LambdaRank but it is based on a family of models called MART (Multiple Additive Regression Trees). These models exploit the ***Gradient Boosted Trees*** that is a cascade of trees, in which the gradients are computed after each new tree, to estimate the direction that minimises the *loss function* (that will be scaled by the contribution of the next tree). In other words, each tree contributes to a *gradient step* in the direction that *minimizes* the loss function. The ensemble of these trees is the final model (*i.e.*, Gradient Boosting Trees). LambdaMART uses this ensemble but it replaces that gradient with the *lambda* (gradient computed given the candidate pairs) presented in LambdaRank.

This algorithm is often considered a ***Pairwise approach*** since the *lambda* considers pairs of candidates, but it actually requires to know the entire ranked list (*i.e.*, scaling the gradient by a factor of the *nDCG* metric, that keeps into account the whole list) – with a clear characteristic of a ***Listwise approach***.

For more details, refer to [Burges et al. 2010] for the description of the three approaches in details or Wellecks' blog posts on Learning to Rank or on LambdaMART for a nice read with great visualizations.

1. Dataset Preparation

Now, going back to our use case the first step, as usual, it's to prepare the data that we need for training LambdaMART.

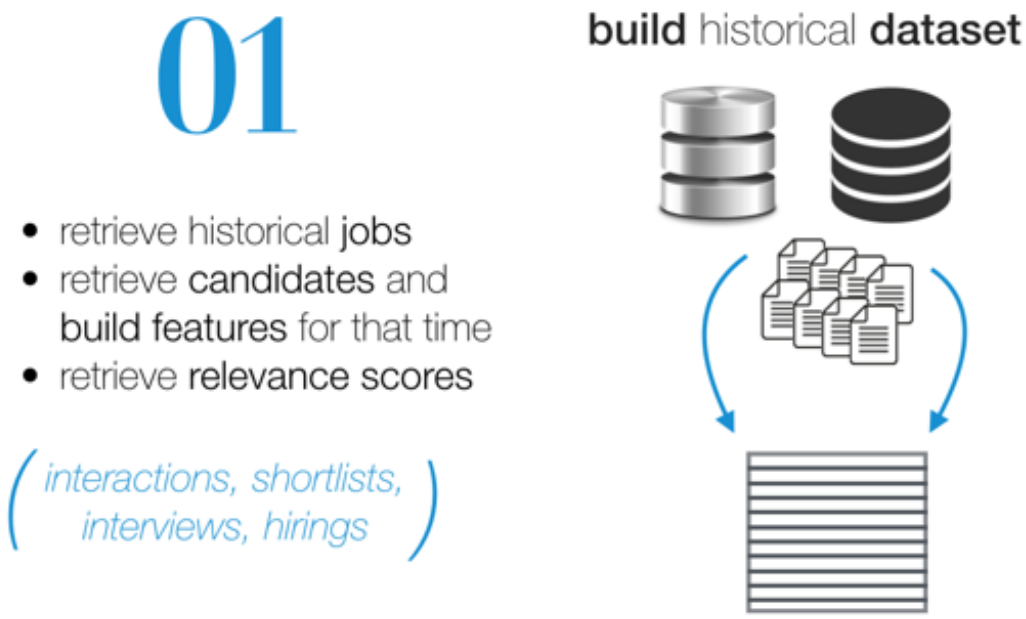

Fig3.

The idea is to reproduce the normal behavior of our Information Retrieval system for a *historical time frame* since we also need the relevance score (so we need to know what happened for each `<job, candidate>` pair). For example, let's assume we want to build the dataset between last *October 1st* and *November 30th*, this means that we have to ***collect all the jobs*** $Q \subseteq Q$ created and ***retrieve all the candidates*** $D \subseteq D$ that were returned in those two months. Then we have also to compute the **relevance scores** $Y \subseteq Y$ for those candidates (*i.e.*, how far they went through the funnel) considering a ***temporal offset*** of a *few weeks*. Jobs created the last week of November need more time to have a comparable set of relevance scores for all its candidates.

In other words, we need to store all the historical data somewhere, in order to build this training set correctly. This is fairly simple when we deal with databases or logs, but it's a bit more problematic when we deal with a search engine such as Elasticsearch. In another blog post, we'll discuss a trick that can be used to maintain (almost effortless) a parallel index that keeps the historical changes of our candidates' profiles.

2. LTR Model Training

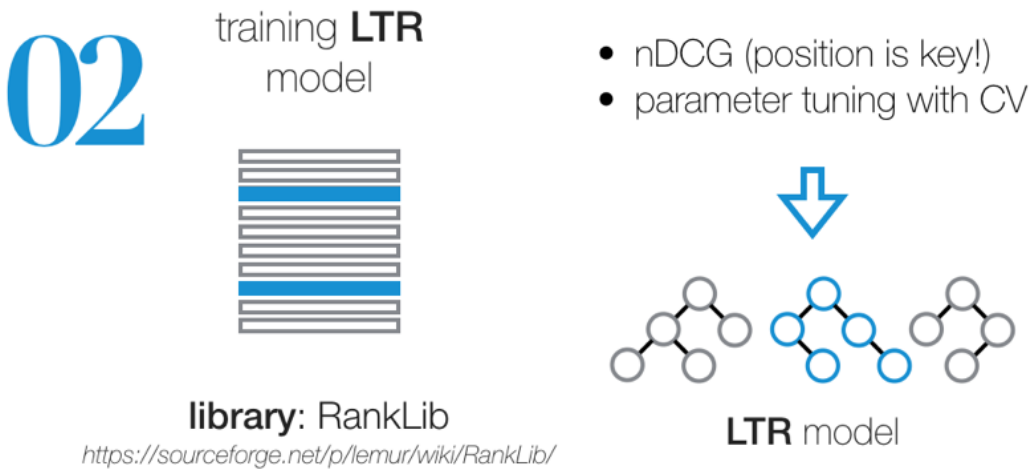

Fig4.

Once we have a historical dataset, we need to train the LambdaMART model using **Cross-Validation (CV)** to perform parameters tuning. We are using **RankLib**, a popular *BSD licensed library* written in Java that includes, among others, implementation of LambdaMART. For alternatives libraries see this Quora Q&A, but considering the maturity and stability level of RankLib, it seems probably the best choice out there.

As evaluation metric, we are using nDCG a very popular ranking metric that is computed *normalizing* the *Discounted Cumulative Gain*.

Very quickly, the ***Cumulative Gain (CG)*** is the sum of the relevance scores of the documents (candidates) retrieved, if we are considering the top `k` positions, it is calculated as the summation of the first `k` relevance scores (see *Fig5/a*). Since we are dealing with a ranking list, a candidate with relevance `y = 5` ranked *fifth* should not count the same as if it was ranked *first*. That's why it was introduced the ***Discounted CG (DCG)***, that basically penalizes the relevance score in the function of the document's position (see *Fig5/b*). However, these scores don't have a fixed ***upper bound*** limit since it depends on the number of results returned by the query, and thus in order to compare DCG of different queries we need to ***normalize them***. To do so, we need to compute the best possible ranking (remember that *we know all the relevance scores* because we are dealing with historical data) and divide the DCG value by this ***Ideal DCG (IDCG)***, obtaining the

nDCG, a value between 0 and 1.

Fig5. Normalized Discount Cumulative Gain (nDCG)

This is why the nDCG is a very good metric for ranking problems. We are using it to evaluate the ranks computed by LambdaMART with different parameter settings, splitting the data with a Cross-Validation fashion.

3. Data and Model Analysis

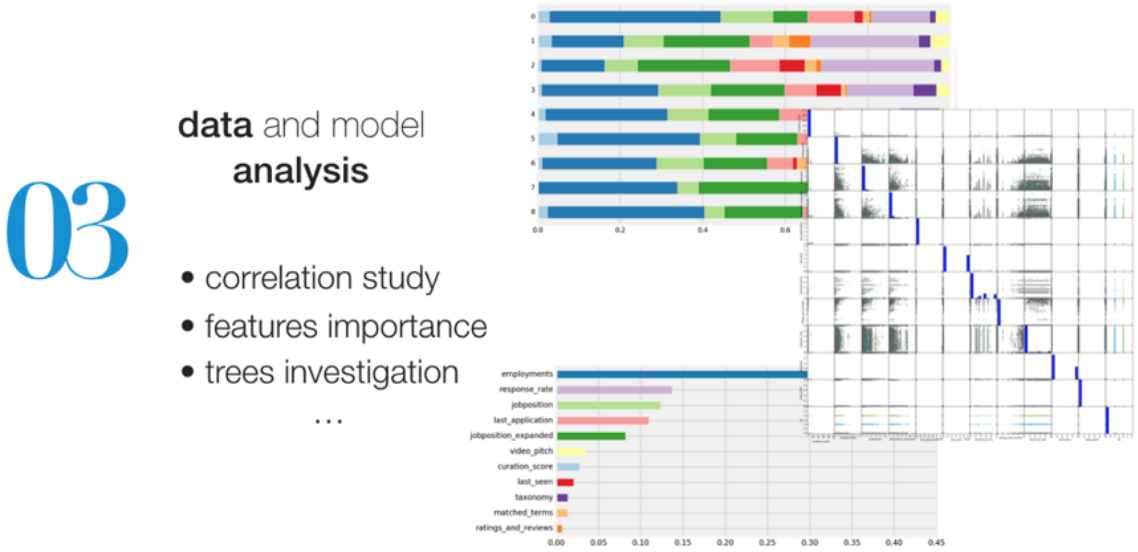

Fig6.

Finally, we have a model trained to compute high-quality rankings given a set of candidates' features. We are almost done, but we should spend some time analyzing the results we've got. A strong advantage of the ***MART*** model, on which is based ***LambdaMART***, is that it allows a deep exploration of the use of the features. Since they are *Trees*, we can investigate how the features are used and, in particular, which is their impact in predicting the right score (*e.g.*, feature importance).

Unfortunately, RankLib doesn't provide any function to *estimate the importance of the features*, and thus we implemented it from scratch using the ***Gini Index*** to estimate the impurity of the split. If you're not familiar with Trees and these

terms, see [Louppe et al. 2014], [Loh 2011], or Jason Brownlee's blog post for further explanation.

This and other types of analysis sheds many lights on the features relation with the **business** and the customer's needs in general, and they can give us important insights on how to improve the **features engineering** step (how to select the features, how to compute them, etc.). A typical workflow includes multiple *iterations of the whole process*, especially when these analyses show strong evidence that adding, removing or changing certain features may lead to improvements in the final ranking.

Conclusions

With this set of three blog posts we have described, on a very high level, the backbone of the ***Information Retrieval framework*** that we used at **Jobandtalent**. As we have seen there are *many possible configurations* and *decisions* that can significantly change the ***quality of the final ranking***. However, with this overview, you should have enough insights to understand how similar problems can be solved.

Soon we are planning to post related articles (more advanced), re-discuss topics with more technical details (e.g., Ensemble of NERs, Query Expansion), or talk about new challenges (*e.g.*, Elasticsearch training index, ETL Framework, Features Builder at scale).

Speeding up Superset by choosing the right database

Data is one of the key ingredients in the success and development of any organization, and here at Jobandtalent, we...

jobandtalent.engineering

We are hiring!

If you want to know more about how is work at Jobandtalent you can read the first impressions of some of our teammates on this blog post or visit our twitter, or check out my personal one.

Thanks to Sebastián Ortega, Sergio Espeja and Ana Freire for feedback and reviews.