



# > Конспект > 4 урок > Особенности работы с деревянными моделями. **YETIRANK**

## > Оглавление

- > [Оглавление](#)
- > [RankNet: проблемы подхода](#)
- > [LambdaRank](#)
- > [MART \(Boosting\)](#)
- > [LambdaMART](#)
- > [YetiRank](#)
- > [Резюме](#)

## > RankNet: проблемы подхода

RankNet, как уже обсуждалось в предыдущей лекции, представляет собой реализацию Pairwise подхода, или подхода, в котором попарно сравниваются предсказания релевантностей для документов. Иными словами, модель занимается **упорядочиванием пар документов** или **минимизацией числа попарных ошибок**.

С одной стороны, в рамках задачи ранжирования это кажется вполне разумным, однако далеко не всегда минимизация попарных ошибок сильно

коррелирует с метриками, которые мы хотим оптимизировать. Данная проблема проиллюстрирована в следующем примере:



Пусть у нас имеются 2 списка из 8 документов для упорядочивания, причём в каждом списке по **2 релевантных** документа. В первом случае они расположены на первой и последней позициях, а во втором — на 3-й и 4-й. Если напрямую рассчитать **количество неупорядоченных пар** в обоих списках, то мы увидим, что во втором случае оно меньше, чем в первом. Но если посмотреть на другие метрики ранжирования, такие как **Average Precision (AP)**, **Discounted Cummulative Gain (DCG)** и **Winner Takes All (WTA)** (бинарная метрика, иллюстрирующая, является ли самый первый документ в упорядоченном списке действительно релевантным), то окажется, что во втором случае они хуже, чем в первом.

Одной из причин, почему **RankNet** приводит к такой деградации метрик, является тот факт, что мы не учитываем важность документов: **некоторые из них куда важнее отранжировать в первую очередь**. В идеале нам бы хотелось, чтобы вес документа зависел от того, как сильно изменится целевая метрика от перестановки конкретного объекта.

Возможные пути решения этой проблемы:

1. Перевзвешивать пары при расчёте функции потерь;
2. Изменять функцию потерь путём внесения информации о целевой метрике;
3. **Менять градиент функции пропорционально целевой метрике.**

Именно последний подход был предложен авторами [LambdaRank](#).

## > LambdaRank

Напомним, что в качестве **Loss-функции**, или функции потерь, в RankNet используется **Cross Entropy (кросс энтропия)**  $C$ . Также напомним некоторые обозначения:

$s_i, s_j$  — предсказания релевантности для  $i$ -го и  $j$ -го документов

$\sigma(x)$  — вообще говоря, любая монотонно возрастающая, положительная функция, а в нашем случае — сигмоида

$S_{ij}$  — новое трансформированное значение целевой переменной (target), получающееся в результате линейного преобразования старых значений (диапазон  $[0, 1] \rightarrow [-1, 1]$ )

Рассмотрим, как выглядят **градиенты функции потерь  $C$  по предсказанным значениям релевантностей  $s_i$** . Можно заметить, что значение совпадает для  $i$ -го и  $j$ -го документа с точностью до знака, что вполне логично: пара документов, которые неправильно отранжированы относительно друг друга, должны "тянуться" в разные стороны с одинаковой "силой" в силу симметричности:

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

Стоит заметить, что в формуле сверху рассматриваются градиенты **относительно предсказаний  $s$** , а не весов самой модели **RankNet**. Поэтому давайте теперь рассмотрим полную формулу уже **относительно весов** нашей модели:

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

В этой формуле фигурирует уже знакомая нам дробь, которую мы рассматривали в формуле выше. Мы можем произвести замену, после чего вынести общую часть за скобки. В результате у нас получается произведение из двух множителей: в первых скобках у нас записано выражение, характеризующее **целевое изменение предсказаний** модели, а во вторых — **градиенты для изменения весов** модели.

Далее для простоты и краткости весь первый множитель этого произведения обозначим как  $\lambda_{ij}$

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

И тогда получим куда более короткую и лаконичную запись формулы для градиентов весов модели:

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

Далее, не ограничивая общности случая, переупорядочим все пары предсказанных релевантностей так, чтобы  $i$ -й документ был всегда более релевантен, чем  $j$ -й. Тогда формула сократится ещё сильнее, ведь  $S_{ij}$  всегда будет равно 1.

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

Ключевая особенность этой формулы в том, что сейчас она зависит **только** от наших подсказаний,  $s_i$  и  $s_j$ , и не зависит от функции потерь, а значит мы больше **не ограничены** в выборе функции для  $\lambda$ . Она может быть сколь угодно сложной функцией. Это позволяет нам обойти трудности, связанные с сортировкой в большинстве метрик для задач ранжирования и IR (information retrieval), и напрямую работать с их оптимизацией, а не ограничиваться корректным упорядочиванием пар.

Тогда формула для обновления весов модели получается следующей:

$$\delta w_k = -\eta \sum_{\{i,j\} \in I} \left( \lambda_{ij} \frac{\partial s_i}{\partial w_k} - \lambda_{ij} \frac{\partial s_j}{\partial w_k} \right) \equiv -\eta \sum_i \lambda_i \frac{\partial s_i}{\partial w_k}$$

Здесь за  $\lambda_i$  обозначена сумма по **всем** парам в датасете, в которые входит  $i$ -й элемент:

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

Здесь очень просто провести аналогию с миром физики. Наш  $i$ -й документ можно представить как некоторую массу. Каждый  $j$ -й документ в первой сумме при вычислении  $\lambda_i$  — это по сути маленькая **сила**, которая подталкивает наш документ **вверх** или вниз в списке выдачи, повышая или понижая его предсказанное значение релевантности. Эту силу можно представить в виде стрелочки (см. картинку ниже).



Сначала берём документы **ниже** нашего  $i$ -го в выдаче, т.е. где он является **первым** в паре (первая сумма в  $\lambda_i$ ). Это будут **"поднимающие" силы**, которые тянут документ **вверх**. Затем наоборот — документы **выше** нашего  $i$ -го в выдаче (вторая сумма в  $\lambda_i$ ). И это **противодействующие** силы, которые **занижают** наше предсказание релевантности. Сумма по всем таким объектам и есть суммарное воздействие на наш объект. Таким образом мы понимаем, как нужно изменять его оценку.

Если же посмотреть, как выбирают значение  $\lambda_i$  в реальной жизни, то на практике значение уже знакомой нам  $\lambda_i$  **домножают на значение изменения целевой метрики**, например **nDCG** (тут она немного модифицирована: вместо **gain** в числителе стоит степенная функция от него).

$$\text{DCG} = \sum_i \frac{2^{\text{rel}_i} - 1}{\log_2(i + 1)}$$

$$\text{nDCG} = \frac{\text{DCG}}{\text{IDCG}}$$

Если же рассмотреть, как изменится метрика от перемены мест  $i$ -го и  $j$ -го документов, то мы получим следующую формулу:

$$\lambda_{ij} = N \left( \frac{1}{1 + e^{s_i - s_j}} \right) (2^{\text{rel}_i} - 2^{\text{rel}_j}) \left( \frac{1}{\log_2(i + 1)} - \frac{1}{\log_2(j + 1)} \right)$$

Грубо говоря, чем **больше прирост метрики** от повышения или понижения позиции в ранжировании, тем **больше это влияет на изменение предсказаний модели**. В итоге более релевантные документы давят сверху, заставляя нерелевантные опускаться вниз, а менее релевантные давят снизу, заставляя всплывать подходящие документы.

Важно отметить, что расчёт градиентов идёт уже **после сортировки документов по оценкам** и градиенты текут как будто бы от нашей метрики, от nDCG или другой. То есть мы сначала берём все документы для запроса, **прогоняем их через модель** и получаем предсказания релевантности каждого отдельного объекта. До этого момента метод напоминает **pointwise**-подход. Далее мы берём все пары из этого ранжирования и пытаемся **оценить изменение метрики при перестановке этих двух объектов из пары**. После этого для каждого документа **рассчитываем суммарную лямбду**, которая говорит о том, куда нам лучше направить изменение предсказаний — вверх или вниз.

Трюк с вынесением  $\lambda_i$  в выражении называется **факторизацией**. За счёт него мы получаем выигрыш в скорости работы, так как он позволяет нам за один прогон (за один forward-pass) по модели получить все предсказания для выдачи, подсчитать  $\lambda_i$  и только после этого делать backward-pass (backpropagation), то есть операцию обратного распространения ошибки для расчёта градиентов. Этот шаг очень дорогой с точки зрения вычислительных мощностей. Если раньше для  $n$  документов мы получали квадратичную зависимость от числа **пар документов** в датасете, то теперь мы используем пары для дешёвого расчёта изменения **nDCG**, а расчёт градиентов происходит **один раз** на каждый документ, на каждую  $\lambda_i$ . Это привело к **очень значительному ускорению** обучения RankNet. Фактически время обучения упало с **почти квадратичного** по количеству релевантных документов на запрос до **почти линейного**.

## > MART (Boosting)

**MART** (Multiple Additive Regression Trees) — алгоритм бустинга регрессионных деревьев.

Рассмотрим **вкратце** алгоритм построения регрессионных **деревьев**:

Пусть в некоторый момент построения мы имеем некоторую выборку объектов с их собственным набором признаков, представленных вектором. Для каждого признака:

1. Переберём все значения этого признака и будем строить разбиение объектов простым условием: значение признака  $j$  меньше некоторого порогового числа  $k$ . Если это условие выполняется, то объект попадает в левое поддерево, если же оно ошибочно — в правое.
2. Затем по всем объектам левого поддерева  $L$  и правого  $R$  мы рассчитываем среднее значение предсказываемого значения  $\mu$ .
3. В каждом из двух множеств считаем среднеквадратичное отклонение (СКО). Общее СКО  $S_j$  считаем как сумму СКО в левом и правом множествах.
4. Затем среди всех перебранных комбинаций разбиений мы выбираем разбиение с наименьшим значением этого среднеквадратичного отклонения  $S_j$ .

Функция ошибки может быть другой, но в классических деревьях для задачи регрессии используется именно СКО.

$$S_j \equiv \sum_{i \in L} (y_i - \mu_L)^2 + \sum_{i \in R} (y_i - \mu_R)^2$$

Далее мы рекурсивно повторяем наш алгоритм для левого и правого поддерева, пока не достигнем некоторого критерия остановки, например максимальной глубины дерева или минимального количества объектов в листе. Для самого последнего уровня дерева, или правильнее сказать для терминальных нод, мы определяем наше предсказание как среднее всех объектов, попавших в лист дерева.

Градиентный бустинг — один из способов построения ансамбля  $F_N(x)$  решающих деревьев. Для его построения мы используем ансамбль последовательно обучаемых деревьев  $f_i(x)$ , предсказание каждого из которых входит в итоговую сумму предсказаний с некоторым весом  $\alpha_i$ . Этот вес определяется каким-нибудь итеративным оптимизационным алгоритмом.

$$F_N(x) = \sum_{i=1}^N \alpha_i f_i(x)$$

Пусть на каком-то этапе у нас есть некоторое количество построенных деревьев. Как же учить следующее? На текущей итерации строится  $m$ -е дерево, когда на вход приходит суммарное предсказание  $m - 1$  предыдущих деревьев.

Теперь нам необходимо понять, какие ответы должны получиться на следующем дереве, чтобы в сумме с текущими ответами наша ошибка минимизировалась. В этом нам поможет градиент, или точнее антиградиент, так как мы решаем задачу минимизации. Мы хотим понять, как нам нужно

изменить наше предсказание на объекте, чтобы минимизировать некоторый функционал  $L$  от целевой переменной  $y_i$  и текущего предсказания  $F_{m-1}(x_i)$ . То есть по сути это как раз ровно такая производная, что указана в формуле ниже — наш антиградиент.

$$\bar{y}_i = - \left[ \frac{\partial L(y_i, F(x_i))}{\partial F(x_i)} \right]_{F(x)=F_{m-1}(x)}$$

Такое новое дерево должно **корректировать уже имеющиеся предсказания**, и потому ему на вход в качестве целевых переменных, в качестве меток, приходят не наши базовые  $y_i$  из выборки, а так называемые "**невязки**" или **остатки** (сколько нам не хватает для того, чтобы дать точное предсказание).

**Обратим внимание, что по сути это в точности наши  $\lambda_i$  из LambdaRank, из метода для задачи ранжирования!**

$$\lambda_i \equiv \sum_{j \in P_i} \frac{\partial C(s_i, s_j)}{\partial s_i}$$

Естественным образом напрашивается объединение алгоритма **LambdaRank** и **MART**. Получается [LambdaMART](#)

## > LambdaMART

Ссылка на статью, в которой подробно описывается происходящее: [ГУГ](#)

Рассмотрим получившийся **алгоритм Lambda[S]MART**:

---

**Algorithm 1** The LambdaSMART algorithm.

---

```
1: for  $i = 0$  to  $N$  do
2:    $F_0(x_i) = \text{BaseModel}(x_i)$  \\ BaseModel may be empty or set to a sub-
   model.
3: end for
4: for  $m = 1$  to  $M$  do
5:   for  $i = 0$  to  $N$  do
6:      $y_i = \lambda_i$ 
7:      $w_i = \frac{\partial y_i}{\partial F(x_i)}$ 
8:   end for
9:    $\{R_{lm}\}_{l=1}^L$  \\ Create  $L$ -terminal node tree on  $\{y_i, x_i\}_{i=1}^N$ 
10:   $\gamma_{lm} = \frac{\sum_{x_i \in R_{lm}} y_i}{\sum_{x_i \in R_{lm}} w_i}$  \\ Find the leaf values based on approximate Newton
    step.
11:   $F_m(x_i) = F_{m-1}(x_i) + v \sum_l \gamma_{lm} 1(x_i \in R_{lm})$ 
12: end for
```

---

Псевдокод алгоритма LambdaSMART

- [2] Итак, сначала для всех  $N$  объектов выборки **получим входные предсказания**. Это могут быть предсказания, например, **ListNet** или **RankNet**.
- [4] Затем мы начинаем цикл постройки  $M$  деревьев.
- [5-6] Снова пройдёмся по всем объектам  $N$  и для каждого **посчитаем наш градиент**. Если при построении классического градиентного бустинга мы использовали производную от функции MSE, которая равняется простой разности величин  $y_i - f(x_i)$ , то **теперь мы считаем**  $\lambda_i$ , как это делалось в **LambdaRank**. На основе текущих предсказаний мы **строим ранжирование**, определяем **прирост метрик от перестановки**, рассматриваем все пары в **выдаче и суммируем**, чтобы получить финальное значение. Напомним, что смысл этой  $\lambda_i$  в том, что она указывает, как и в каком направлении нужно изменить наше предсказание, чтобы оптимизировать функцию ошибки или улучшить метрики.
- [7] Тут же рассчитывается  $w_i$  — вторая производная, **используемая для расчёта веса при суммировании**. Её мы опустим, так как к ранжированию это не имеет никакого отношения, это **компонент градиентного бустинга**.
- [9] После того, как мы посчитали все  $\lambda_i$ , мы **их объявляем нашими целевыми переменными** для построения нового дерева, и **строим его для**

наших объектов  $\{x_i\}_{i=1}^N$  с целью предсказать  $\lambda_i$ , то есть новый  $y_i$ . Если в классическом градиентном бустинге в задаче регрессии мы брали разницу целевого и предсказанного значения, то сейчас мы пытаемся обучиться, используя  $\lambda_i$  в качестве целевой переменной, которая при построении каждого нового дерева меняется, так как меняются предсказанные значения. Эта замена производных работает потому, что и  $\lambda_i$ , и разность указывают нам на то, как именно нужно поменять предсказываемые релевантности, чтобы уменьшить значение функции потерь.

- [10] Далее рассчитываем длину нашего градиентного шага, которая уникальна для каждого отдельного листа. Это не learning rate, это отдельный коэффициент. Он вычисляется с помощью метода Ньютона (детали мы опустим). Суть в том, что просто подбирается такое число, при умножении на которое предсказаний нового построенного дерева мы, насколько это возможно, минимизируем функцию потерь.
- [11] И наконец, мы обновляем наш ансамбль, добавляя к уже полученным скорам предсказания нового дерева. При этом здесь есть и learning rate, который обозначен как  $v$  и обычно одинаков для всех деревьев в бустинге, и наш коэффициент, рассчитанный для каждого отдельного листа каждого отдельного дерева.

## > YetiRank

Ссылка на оригинальную статью для интересующихся: [тут](#)

Сейчас YetiRank широко используется, например, в CatBoost'е — реализации градиентного бустинга от Yandex. Разработан сам метод тоже сотрудниками Яндекса. На одной из страниц документации к CatBoost'у указывается, что он один из самых качественных с точки зрения средних метрик на разных датасетах. Однако он достаточно медленный. Интересно, что именно YetiRank выиграл те же соревнования, что и LambdaMART, но годом позже — в 2011 году.

Итак, в качестве основной функции потерь используется та же самая формула, что была и в RankNet при pairwise-подходе, однако каждая пара дополнительно взвешивается множителем  $w_{ij}$ . Авторы также упоминают возможность добавления уже изученной нами  $\lambda_i$  при расчёте градиентов, но мы абстрагируемся от этого и сделаем упор на ключевых изменениях.

$$\mathbb{L} = - \sum_{(i,j)} w_{ij} \log \frac{e^{x_i}}{e^{x_i} + e^{x_j}}$$

$$w_{ij} = N_{ij} c(l_i, l_j)$$

Каждый из весов унекален для конкретной пары документов при обучении и состоит из двух компонент, двух сомножителей. Первый множитель  $N_{ij}$  помогает нам при обучении понять, насколько данная пара важна для ранжирования. Второй множитель — некоторая функция отметок релевантности  $l_i$  и  $l_j$  из разметки. Она показывает нам, насколько легко эти метки перепутать.

Давайте внимательно рассмотрим первую составляющую. Как уже обсуждалось, не все объекты важны при ранжировании, и иногда лучше определённый документ поднимать со дна выдачи. Для оценки важности предлагается на очередной итерации бустинга брать предсказываемые значения  $x_i$  и добавлять к ним шум по указанной ниже формуле:

$$\hat{x}_i = x_i + \log \frac{r_i}{1 - r_i}$$

Этот шум может быть положительным или отрицательным, а  $r_i$  принимает значение от 0 до 1 из семплируется из равномерного распределения. Затем производится переранжирование согласно предсказаниям с шумом, за счет чего мы добиваемся некоторого перемешивания.

Далее мы идём сверху вниз, рассматриваем каждые два подряд идущих документа и увеличиваем суммарный вес этой пары на величину метрики MRR, которая считается согласно указанной ниже формуле:

$$N_{ij} = \frac{1}{n} \sum_{t=1}^n \frac{1}{index_t(\min(i, j))}$$

Об этой метрике мы говорили на второй лекции. Напомним, что по сути это просто величина, обратная рангу. В паре мы берём **самый верхний документ**, поэтому нам нужен **минимальный индекс**, что и записано в знаменателе. После переупорядочивания выборки с шумом в предсказаниях мы берём **первый и второй документы** и вес этой пары увеличиваем на единицу, так как среднеобратный ранг будет 1. Потом мы берём второй и третий документы и получаем вес пары 1/2. Выходит, что у нас есть только **веса для тех пар, которые попадались при перемешивании последовательно**. Веса остальных пар равны нулю.

Процедура внесения шума, перемешивания и расчёта веса **повторяется 100 раз**, каждый раз у нас получается другая перестановка. Чем выше каждый раз попадает документ, тем больше его вес, и наоборот. Если у нас за эти 100 попыток **ни разу не шли подряд какие-либо два документа**, а в разметке в

датасете у нас для них есть оценка релевантности, то эта пара **не будет использоваться при обучении на данной итерации**, так как её вес нулевой.

Понятно, что в самом начале обучения вес  $N_{ij}$  практически ни на что не влияет, и он вообще случайный. Однако он **начинает играть существенную роль**, когда в бустинге уже есть несколько построенных деревьев, и **наши предсказания обретают смысл, а получаемое ранжирование неслучайно**.

**Другая составляющая весов** в функции потерь — это вот такая хитрая формула:

$$c(l_i, l_j) = \sum_{u,v} 1_{u>v} p(u|l_i) p(v|l_j)$$

Она помогает нам уделять больше внимания тем парам объектов, у которых **нет неопределённости в проставленной оценке релевантности**, то есть во входных данных. Здесь  $u$  и  $v$  — это конкретные оценки для  $i$ -го и  $j$ -го документов, например от 1 до 5, где 1 означает **нерелевантный мусор**, а 5 — **идеальное соответствие**.

Далее мы пробегаемся по всем возможным комбинациям этих оценок и берём только такие, где **релевантность первого документа выше, чем второго**, так как у нас в датасете изначально сделан тот же трюк, что использовался в **LambdaRANK** с перестановкой пар. Для этих комбинаций происходит умножение единицы на две вероятности, характеризующие возможность того, что **оценка релевантности в датасете на самом деле выставлена неправильно** (люди, размещающие данные, не идеальны и могут ошибаться). Стоит также отметить, что этот множитель у веса пары, а именно функция  $c$  от лейблов  $l_i$  и  $l_j$  не зависит от текущей модели и может быть предрасчитана, так как мы **используем только метки из датасета, но не предсказания**.

Последнее, что хочется обсудить сегодня — как **определять “истинные” значения меток** в разметке, которые были упомянуты выше, а также как строить такую матрицу на основе своих данных. На самом деле метки не являются истинными, это просто **наше предположение** о том, какими они должны быть. Для того чтобы выдвинуть такое предположение, все документы в уже векторизованном виде **разобъём** по так называемым “бакетам”, то есть просто на несколько кластеров. **Объединяем объекты в один кластер**, если расстояние от текущего  $i$ -го объекта **не больше**, чем некоторый маленький  $\epsilon$  до всех других объектов кластера. В идеальном случае у нас в один “бакет” должны попадать объекты с одинаковым набором признаков, то есть идентичные документы, но если говорить в целом, то они просто должны быть **максимально похожи**.

Для документов из "бакета" мы можем посмотреть на распределение меток **релевантности** в разметке, и определить превалирующую составляющую. В статье предлагается просто случайно брать метку из этого кластера, взвешивая вероятность взятия на количество объектов с этой меткой в "бакете". Если все объекты в кластере максимально похожих объектов имеют одинаковую метку, то это значит, что никакой неопределенности в этом кластере нет. А вот если из 20 объектов 17 имеют отметку "хорошее соответствие", а 3 — "идеальное" (и это при том, что с точки зрения признакового описания эти объекты максимально похожи), то скорее всего тут возникла какая-то путаница в разметке. А для построения самой **confusion матрицы** нам подойдёт какой-нибудь **итеративный метод**, оптимизирующий **функцию правдоподобия**, то есть можно просто пытаться сделать значения в матрице максимально похожими на ту картину, что мы наблюдаем в "бакетах".

## > Резюме

- На примере мы разобрались, почему **не всегда нужно оптимизировать количество неупорядоченных пар** и чем это может обернуться.
- Далее мы решили эту проблему с помощью **подмены градиентов**, реализованной в методе LambdaRank. Новые градиенты зависят напрямую от **изменения целевой метрики**, а не от функции потерь. Приём подмены градиентов, как оказалось, можно и нужно использовать в деревянных моделях, а именно в градиентном бустинге. На очередной итерации обучения нового дерева нам нужно в качестве целевых значений указать  $\lambda$ , и тогда модель покажет хорошее качество в задаче ранжирования, вобрав в себя плюсы разобранных выше подходов. Однако и этот подход не лишен минусов, и для их разрешения мы вводим в функцию потерь **веса**, **зависящие от важности объекта и от качества нашей разметки**.
- Всё это предлагает нам такой метод, как **YetiRank**.