Бустинг и ранжирование

Сергей Николенко

НИУ ВШЭ — Санкт-Петербург

18 апреля 2020 г.

Random facts:

- 18 апреля в России — День воинской славы в честь победы над псами-рыцарями на Чудском озере; значение битвы современными историками оспаривается, но чуди точно «паде бещисла»
- 18 апреля 1775 г. Пол Ревир переплыл на лодке устье реки Чарльз, проскользнув мимо HMS Somerset, и поскакал к Лексингтону, куда к полуночи успешно добрался
- 18 апреля 1846 г. Роял Хаус (именно так, Royal E. House) запатентовал телеграфный аппарат
- 18 апреля 1870 г. I Ватиканский собор принял догмат о непогрешимости (точнее, безошибочности, грех здесь ни при чём) Папы Римского
- 18 апреля 1922 г. в Москве была основана футбольная команда МКС, ныне известная как «Спартак», а 18 апреля 1923 г. по инициативе Феликса Дзержинского было создано спортивное общество «Динамо»
- 18 апреля 1928 г. в газете «Правда» вышла статья Максима Горького, в которой джаз назван «музыкой толстых»

Бустинг: AdaBoost

AdaBoost

- Следующая идея объединения моделей: предположим, что у нас есть возможность обучать какую-нибудь простую модель (weak learner) на подмножестве данных.
- Тогда можно делать так: обучили модель, посмотрели, где она хорошо работает, обучили следующую модель на том подмножестве, где она работает плохо, повторили.
- Этот метод называется *бустинг* (boosting).

AdaBoost

- AdaBoost: самый простой вариант. Рассмотрим задачу бинарной классификации; данные – это $\mathbf{x}_1, \ldots, \mathbf{x}_N$ с ответами $t_1, \ldots, t_N$, $t_i \in \{-1, 1\}$.
- Снабдим каждый тестовый пример весом w_i; изначально положим $w_i = \frac{1}{N}$.
- Предположим, что у нас есть процедура, которая обучает некоторый классификатор, выдающий $y(\mathbf{x}) \in \{-1, 1\}$, на *взвешенных* данных (минимизируя взвешенную ошибку).

AdaBoost

- Тогда в алгоритме AdaBoost мы инициализируем $w_n^{(1)} := 1/N$, а потом для $m = 1..M$:
  1. обучаем классификатор $y_m(\mathbf{x})$, который минимизирует функцию ошибки

$$J_m = \sum_{n=1}^{N} w_n^{(m)} \left[y_m(\mathbf{x}_n) \neq t_n \right];$$

  2. вычисляем

$$\epsilon_m = \frac{\sum_{n=1}^{N} w_n^{(m)} \left[y_m(\mathbf{x}_n) \neq t_n \right]}{\sum_{n=1}^{N} w_n^{(m)}}, \quad \alpha_m = \ln\left(\frac{1 - \epsilon_m}{\epsilon_m} \right);$$

  3. пересчитываем новые веса

$$w_n^{(m+1)} = w_n^{(m)} e^{\alpha_m [y_m(\mathbf{x}_n) \neq t_n]}.$$

- После обучения предсказываем как $Y_M(\mathbf{x}) = \operatorname{sign}\left(\sum_{m=1}^{M} \alpha_m y_m(\mathbf{x}) \right)$.

AdaBoost

- Смысл именно такой, как мы говорили: сначала тренируем абстрактно лучший классификатор. Потом увеличиваем веса неправильно классифицированным примерам, обучаем новый классификатор, и т.д.

Теоретические свойства AdaBoost

- Изначально, когда AdaBoost придумали [Freund, Shapire, 1997], мотивация была такая: предположим, что ошибка каждого слабого классификатора h_t не превышает $\epsilon_t = \frac{1}{2} - \gamma_t$.
- Тогда можно показать, что окончательная ошибка не превосходит

$$\prod_t \left(2\sqrt{\epsilon_t(1-\epsilon_t)}\right) = \prod_t \sqrt{1-4\gamma_t^2} \leq e^{-2\sum_t \gamma_t^2}.$$

- Однако на самом деле гарантий на γ_t обычно нету, и практические результаты AdaBoost лучше, чем можно было бы ожидать из этой оценки.

Теоретические свойства AdaBoost

- Основная идея [Friedman et al., 2000]: давайте определим экспоненциальную ошибку

$$E = \sum_{n=1}^{N} e^{-t_n f_m(\mathbf{x}_n)},$$

где f_m – линейная комбинация базовых классификаторов:

$$f_m(\mathbf{x}) = \frac{1}{2} \sum_{l=1}^{m} \alpha_l y_l(\mathbf{x}).$$

- Мы хотим минимизировать E по α_l и параметрам $y_l(\mathbf{x})$.

Теоретические свойства AdaBoost

- Минимизируем $E = \sum_{n=1}^{N} e^{-t_n f_m(\mathbf{x}_n)}$.
- Вместо глобальной оптимизации будем действовать жадно: пусть $y_1(\mathbf{x}), \ldots, y_{m-1}(\mathbf{x})$ и $\alpha_1, \ldots, \alpha_{m-1}$ уже зафиксированы. Тогда ошибка получается

$$E = \sum_{n=1}^{N} e^{-t_n f_{m-1}(\mathbf{x}_n) - \frac{1}{2} t_n \alpha_m y_m(\mathbf{x})} = \sum_{n=1}^{N} w_n^{(m)} e^{-\frac{1}{2} t_n \alpha_m y_m(\mathbf{x})},$$

где $w_N^{(m)} = e^{-t_n f_{m-1}(\mathbf{x}_n)}$ – это как раз и есть наши веса, и их теперь можно считать константами.

Теоретические свойства AdaBoost

- На правильных классификациях произведение -1, на неправильных $+1$:

$$E = e^{-\frac{\alpha_m}{2}} \sum_{\text{correct}} w_n^{(m)} + e^{\frac{\alpha_m}{2}} \sum_{\text{wrong}} w_n^{(m)} =$$

$$= \left(e^{\frac{\alpha_m}{2}} - e^{-\frac{\alpha_m}{2}}\right) \sum_{n=1}^{N} w_n^{(m)} \left[y_m(\mathbf{x}_n) \neq t_n\right] + e^{-\frac{\alpha_m}{2}} \sum_{n=1}^{N} w_n^{(m)},$$

и достаточно минимизировать $J_m = \sum_{n=1}^{N} w_n^{(m)} \left[y_m(\mathbf{x}_n) \neq t_n\right]$.

Теоретические свойства AdaBoost

- Ну а когда мы обучим $y_m(\mathbf{x})$, из $E = \sum_{n=1}^{N} w_n^{(m)} e^{-\frac{1}{2} t_n \alpha_m y_m(\mathbf{x})}$ получится

$$w_n^{(m+1)} = w_n^{(m)} e^{-\frac{1}{2} t_n \alpha_m y_m(\mathbf{x}_n)} = w_n^{(m)} e^{-\frac{1}{2}\alpha_m} e^{\alpha_m [y_m(\mathbf{x}_n) \neq t_n]},$$

  и на $e^{-\frac{1}{2}\alpha_m}$ можно все веса сократить.

- Таким образом, бустинг можно рассматривать как оптимизацию экспоненциальной ошибки.

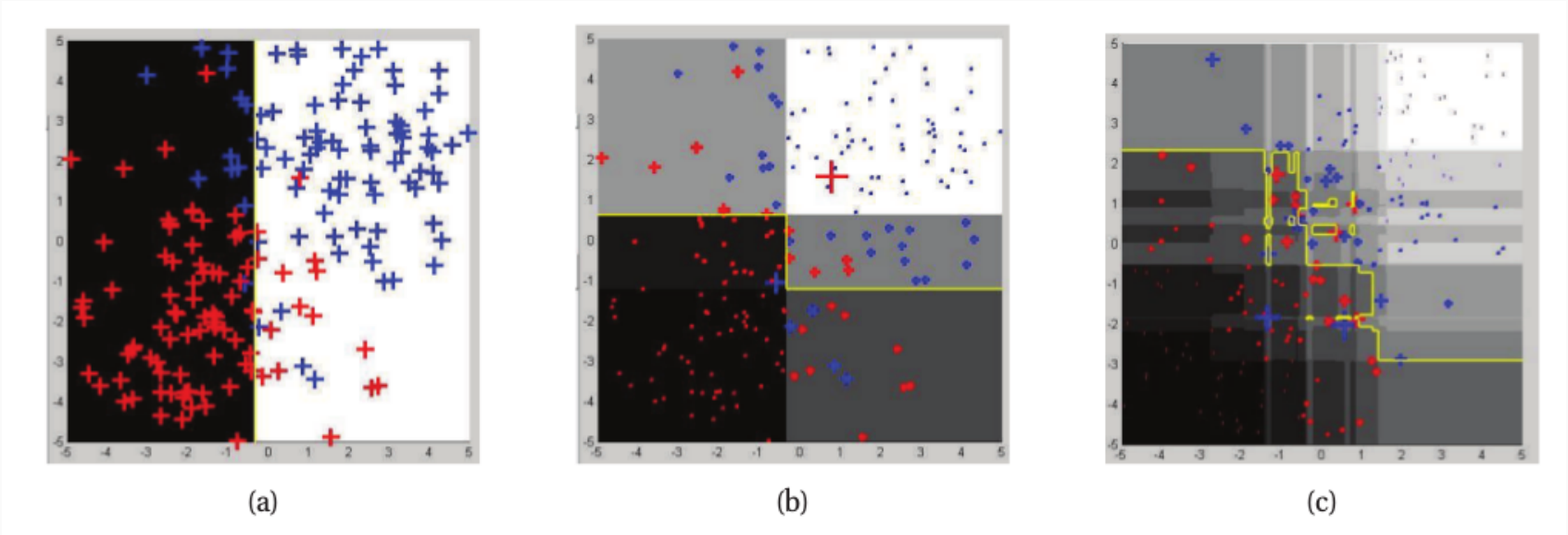
Деревья принятия решений

Weak learners: decision trees

- А что за weak learners применяются в реальных приложениях?
- Обычно бустинг применяется, когда есть набор уже посчитанных фич (посчитанных из каких-то более сложных моделей), и нужно объединить их в единую модель.
- Часто слабые классификаторы очень, очень простые.
- *Пни принятия решений* (decision stumps): берём одну координату и ищем по ней оптимальное разбиение.

Weak learners: decision trees

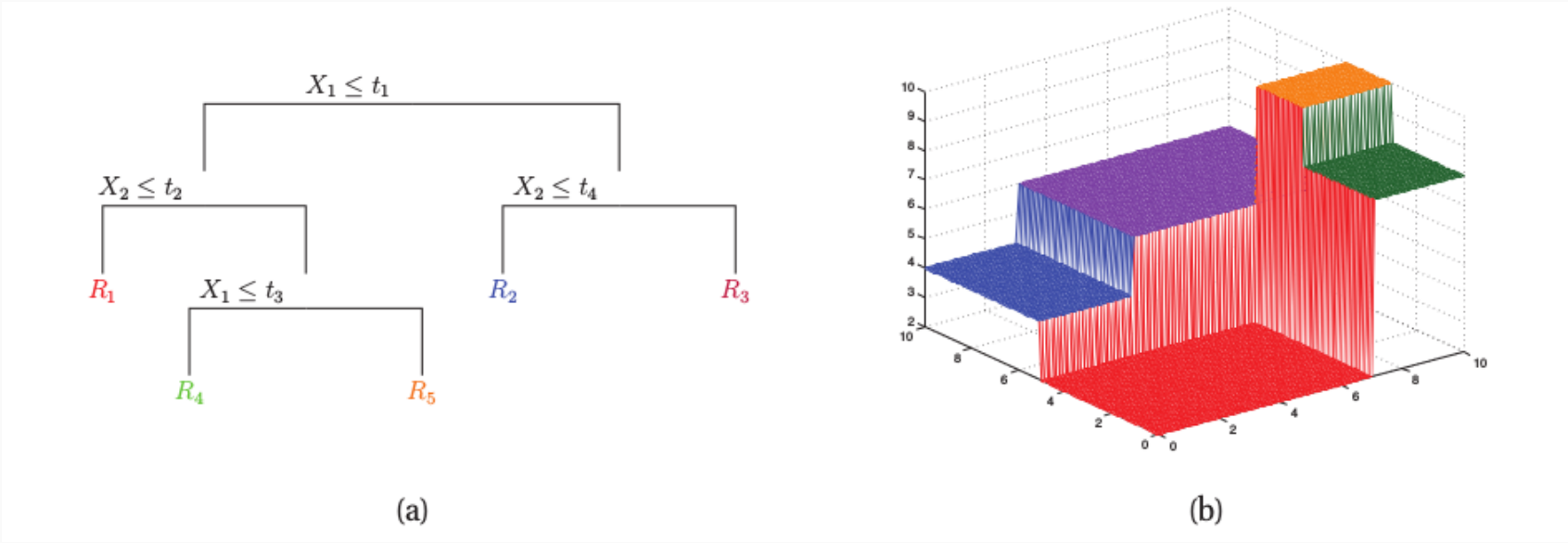
- Пример бустинга на пнях:



- Могут быть чуть посложнее: *деревья принятия решений* (decision trees).

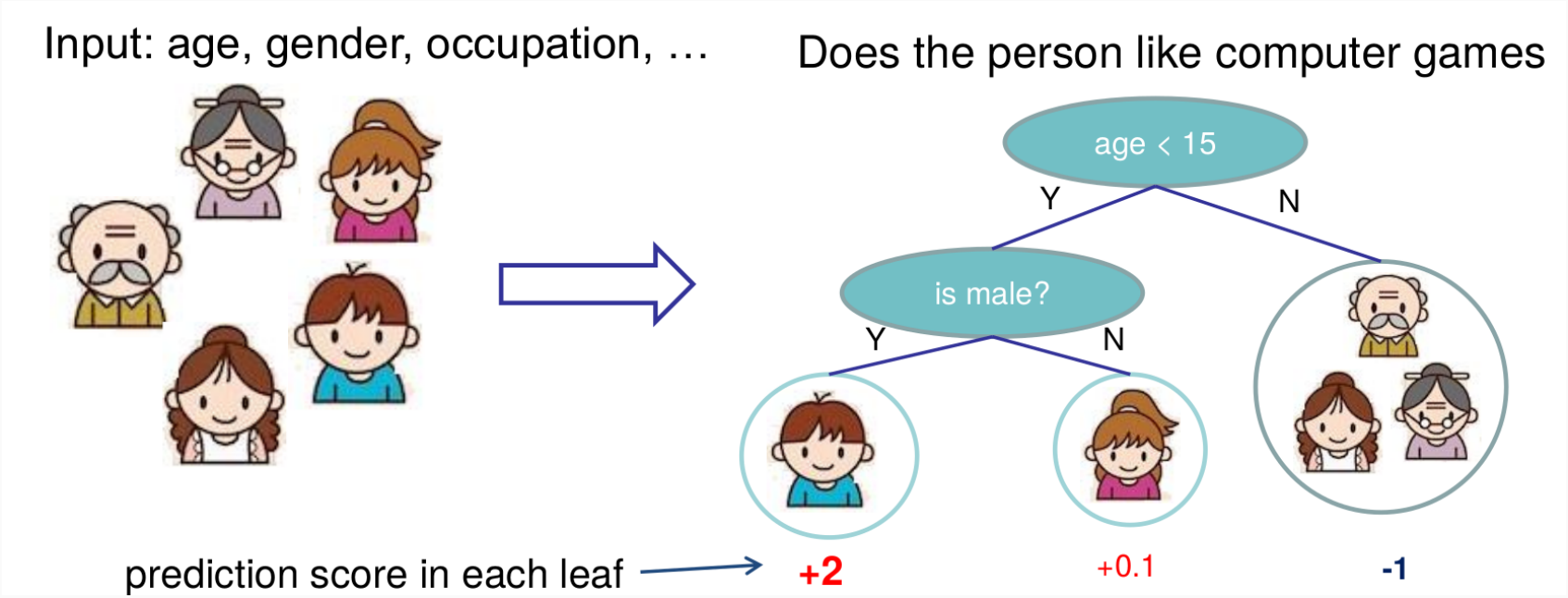
Weak learners: decision trees

- Дерево принятия решений — это дерево. На нём есть метки:
  - в узлах, не являющиеся листьями: атрибуты (фичи), по которым различаются случаи;
  - в листьях: значения целевой функции;
  - на рёбрах: значения атрибута, из которого исходит ребро.
- Чтобы классифицировать новый случай, нужно спуститься по дереву до листа и выдать соответствующее значение.



Weak learners: decision trees

- Картинки от Tianqi Chen и Carlos Guestrin, авторов XGBoost:



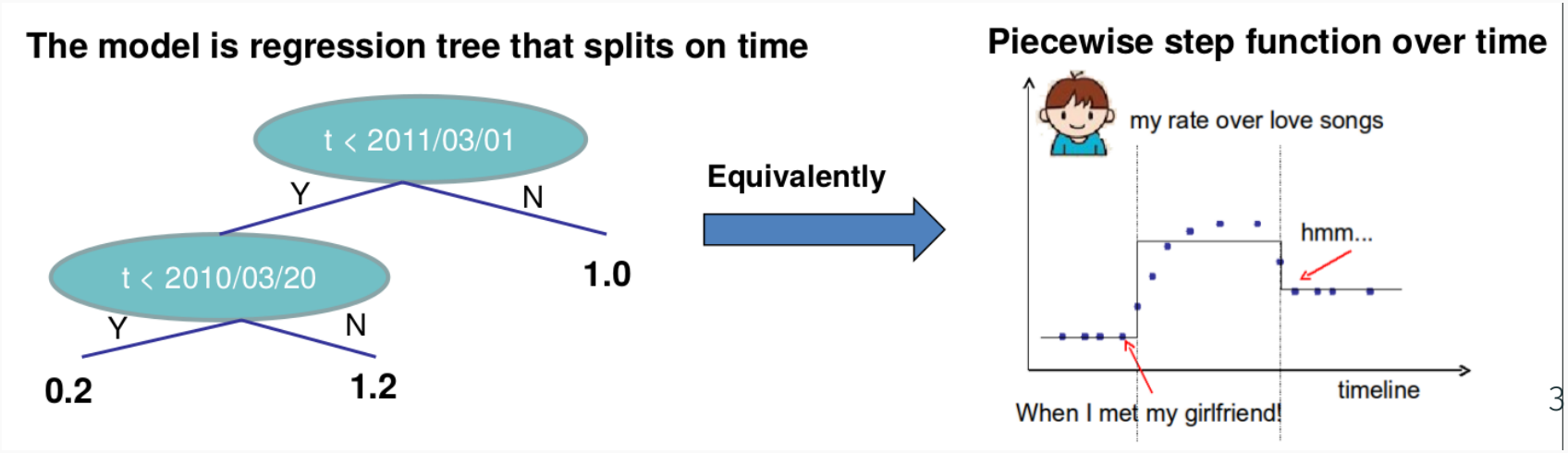
Weak learners: decision trees

- Конечно, перебрать все деревья нельзя, их строят жадно.
  1. Выбираем очередной атрибут Q, помещаем его в корень.
  2. Выбираем оптимальное разбиение по атрибуту. Для всех интервалов разбиения:
     - оставляем из тестовых примеров только те, у которых значение атрибута Q попало в этот интервал;
     - рекурсивно строим дерево в этом потомке.
- Остались три вопроса:
  1. как проводить разбиение?
  2. как выбирать новый атрибут?
  3. когда останавливаться?

Weak learners: decision trees

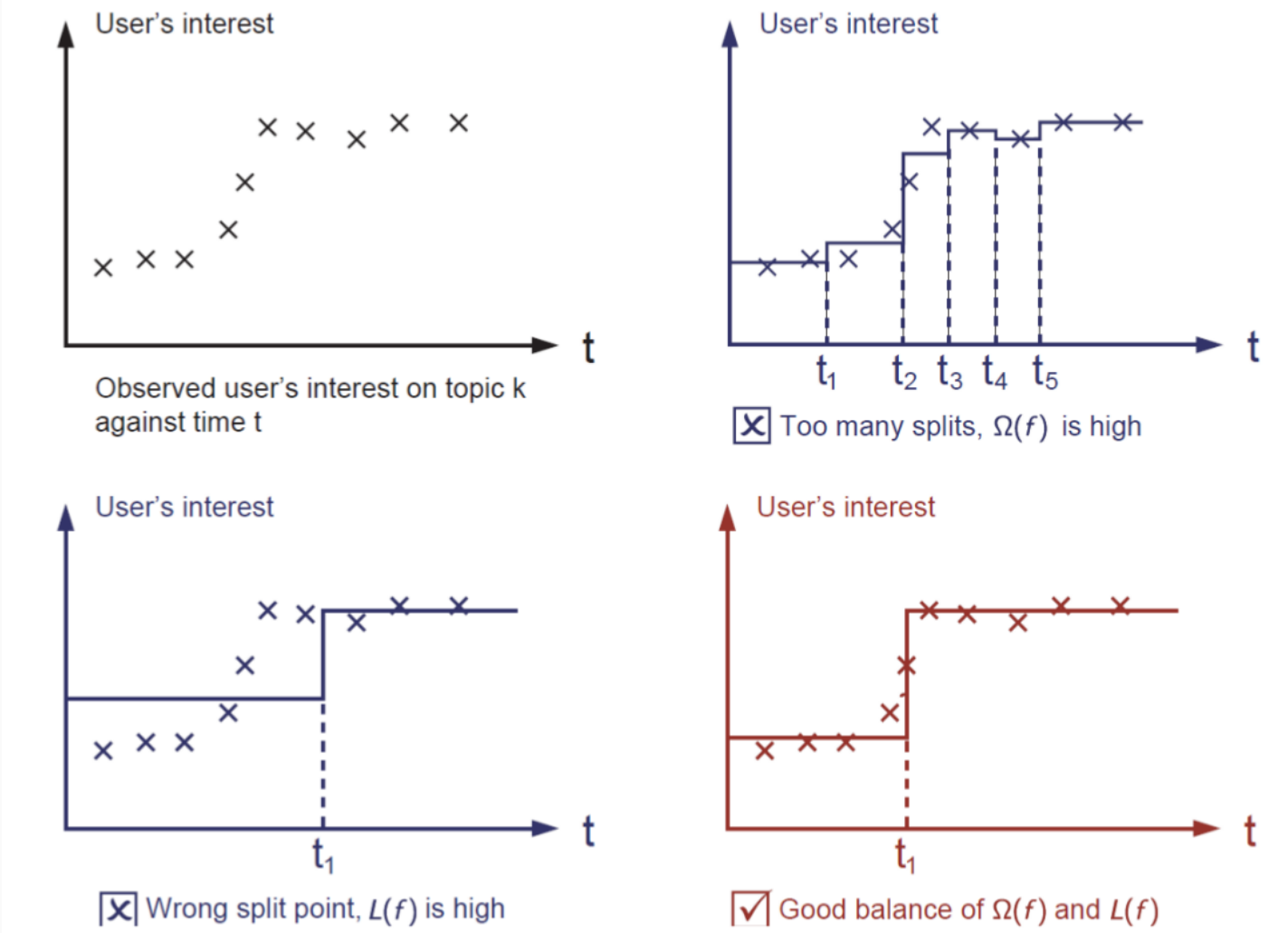
- Если атрибут бинарный или дискретный с небольшим числом значений, то просто по значениям.
- Если непрерывный – можно брать среднее арифметическое (тем самым минимизируя сумму квадратов).
- Выбирают атрибут, оптимизируя целевую функцию. Для задачи регрессии просто минимизируем среднеквадратическую ошибку по отношению к текущему предсказателю

$$y_\tau = \frac{1}{|X_\tau|} \sum_{x_n \in X_\tau} t_n.$$



Weak learners: decision trees

- Как обучить дерево:



Weak learners: decision trees

- Предположим, что мы решаем задачу классификации на K классов.
- Тогда «сложность» подмножества данных X_τ относительно целевой функции $f(\mathsf{x}) : \mathsf{X} \to \{1, \ldots, K\}$ характеризуется *перекрёстной энтропией*:

$$Q(\mathsf{X}_\tau) = \sum_{k=1}^{K} p_{\tau,k} \ln p_{\tau,k}.$$

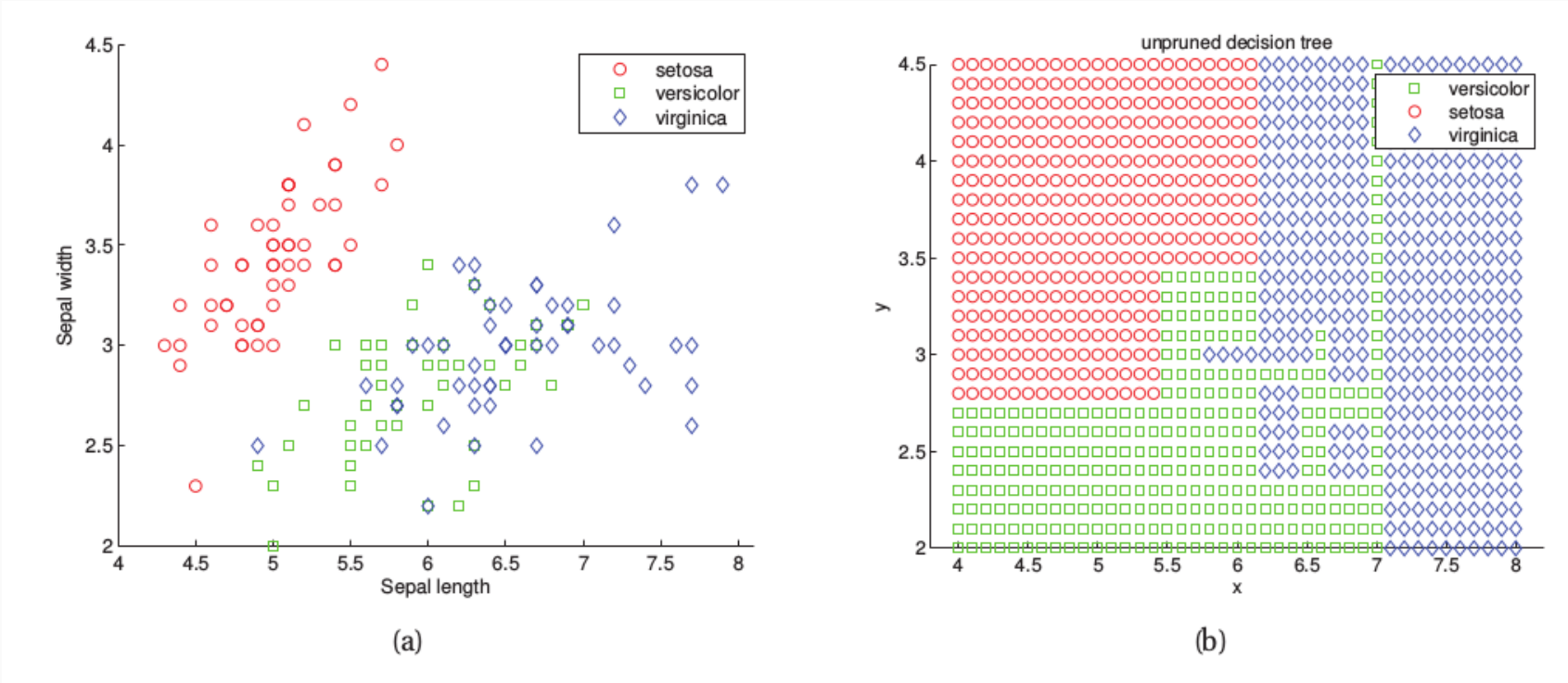
- Иногда ещё используют *индекс Джини* (Gini index): $G(\mathsf{X}_\tau) = \sum_{k=1}^{K} p_{\tau,k}(1 - p_{\tau,k})$.

Weak learners: decision trees

- Когда останавливаться? Недоучиться плохо и переучиться плохо.
- Останавливаться, когда ошибка перестанет меняться, тоже плохо (она может опять начать меняться ниже).
- Поэтому делают так: выращивают большое дерево, чтобы наверняка, а потом *обрезают* его (pruning): поддерево τ схлопывают в корень, а правило предсказания в корне считают как $y_\tau = \frac{1}{|\mathsf{X}_\tau|} \sum_{\mathsf{x}_n \in \mathsf{X}_\tau} t_n$.
- Обрезают, оптимизируя функцию ошибки с регуляризатором: $\sum_{\tau=1}^{|T|} Q(\mathsf{X}_\tau) + \lambda |T|$ (для классификаторов здесь можно использовать долю ошибок классификации).

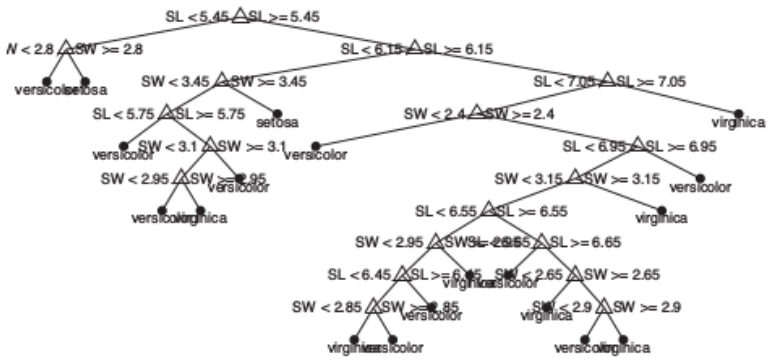
Пример

- Пример на датасете iris.
- Вход:

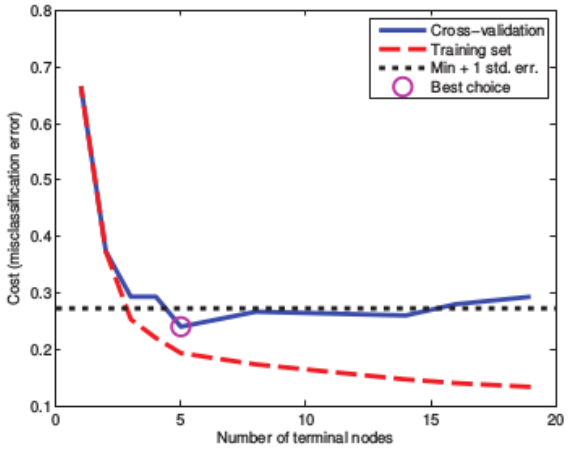


Пример

- Слишком глубокое дерево и его ошибки:



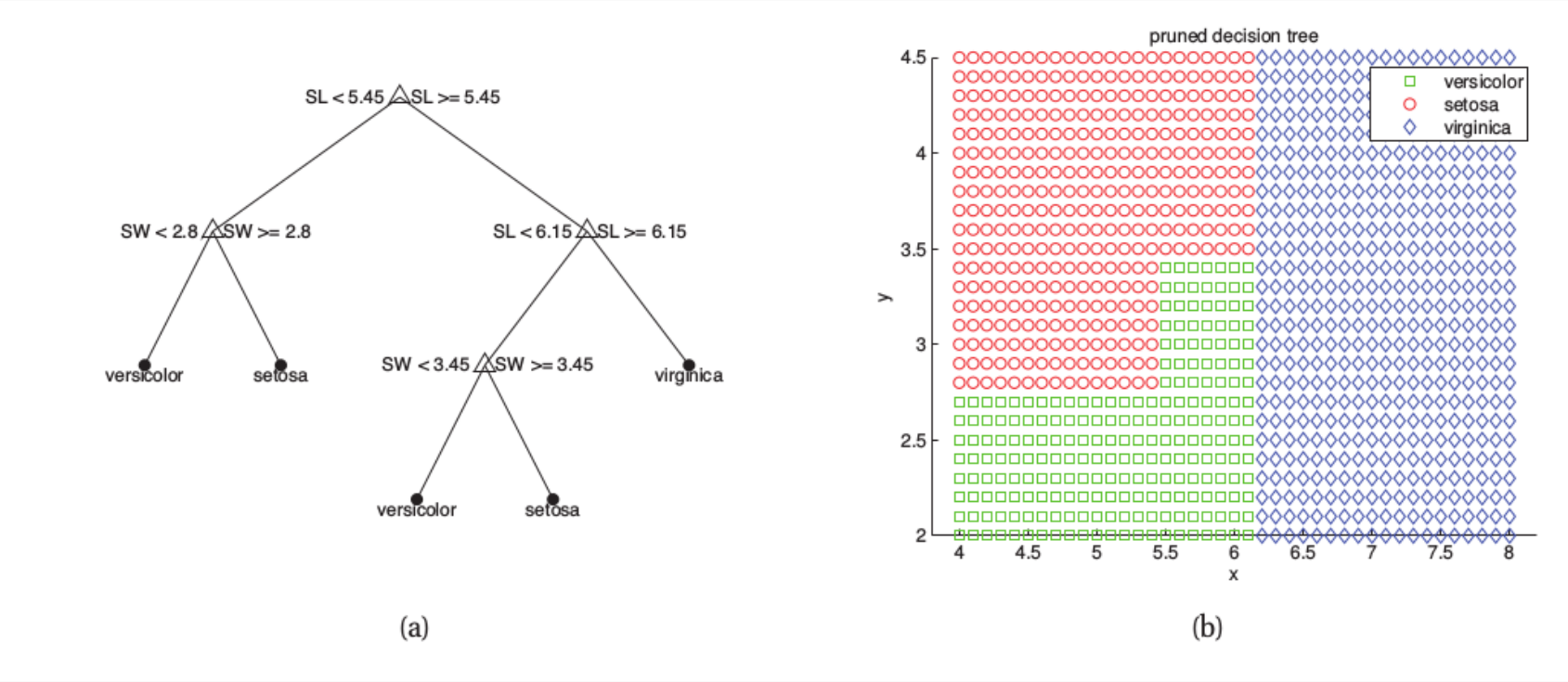
(a)



(b)

Пример

- Обрезанное дерево:



Градиентный бустинг

Градиентный бустинг

- Теперь – к градиентному бустингу (xgboost – это как раз градиентный бустинг).
- Предположим, что мы хотим обучить ансамбль из K деревьев:

$$\hat{y}_i = \sum_{k=1}^{K} f_k(x_i), \text{ где } f_k \in \mathcal{F}.$$

- Целевая функция – это потери + регуляризаторы:

$$\text{Obj} = \sum_{i=1}^{N} l(y_i, \hat{y}_i) + \sum_{k=1}^{K} \Omega(f_k).$$

- Например, для регрессии $l(y_i, \hat{y}_i) = (y_i - \hat{y}_i)^2$.
- А для классификации в AdaBoost было

$$l(y_i, \hat{y}_i) = y_i \ln(1 + e^{-\hat{y}_i}) + (1 - y_i) \ln(1 + e^{\hat{y}_i}).$$

Градиентный бустинг

- Мы не можем просто взять и минимизировать общую ошибку – трудно минимизировать по всевозможным деревьям.
- Так что опять продолжаем жадным образом:

$$\begin{aligned}
\hat{y}_i^{(0)} &= 0, \\
\hat{y}_i^{(1)} &= f_1(x_i) = \hat{y}_i^{(0)} + f_1(x_i), \\
\hat{y}_i^{(2)} &= f_1(x_i) + f_2(x_i) = \hat{y}_i^{(1)} + f_1(x_2), \\
&\dots,
\end{aligned}$$

а предыдущие деревья всегда остаются теми же самыми, они фиксированы.

Градиентный бустинг

- Чтобы добавить следующее дерево, нужно оптимизировать

$$\hat{y}_i^{(t)} = \hat{y}_i^{(t-1)} + f_t(x_i), \text{ так что}$$

$$\mathrm{Obj}^{(t)} = \sum_{i=1}^{N} l\left(y_i, \hat{y}_i^{(t-1)} + f_t(x_i)\right) + \Omega(f_t) + \mathrm{Const.}$$

- Например, для квадратов отклонений

$$\mathrm{Obj}^{(t)} = \sum_{i=1}^{N} \left(2(\hat{y}_i^{(t-1)} - y_i) f_t(x_i) + f_t(x_i)^2\right) + \Omega(f_t) + \mathrm{Const.}$$

Градиентный бустинг

- Чтобы оптимизировать

$$\text{Obj}^{(t)} = \sum_{i=1}^{N} l(y_i, \hat{y}_i^{(t-1)} + f_t(x_i)) + \Omega(f_t) + \text{Const},$$

давайте заменим это на аппроксимацию второго порядка.

- Обозначим

$$g_i = \frac{\partial l(y_i, \hat{y}^{(t-1)})}{\partial \hat{y}^{(t-1)}}, \quad h_i = \frac{\partial^2 l(y_i, \hat{y}^{(t-1)})}{\partial (\hat{y}^{(t-1)})^2},$$

тогда

$$\text{Obj}^{(t)} \approx \sum_{i=1}^{N} \left(l(y_i, \hat{y}_i^{(t-1)}) + g_i f_t(x_i) + \frac{1}{2} h_i f_t^2(x_i) \right) + \Omega(f_t) + \text{Const}.$$

- Например, для квадрата отклонения $g_i = 2(\hat{y}^{(t-1)} - y_i)$, $h_i = 2$.

Градиентный бустинг

- Итак,

$$\text{Obj}^{(t)} \approx \sum_{i=1}^{N} \left(l(y_i, \hat{y}_i^{(t-1)}) + g_i f_t(x_i) + \frac{1}{2} h_i f_t^2(x_i) \right) + \Omega(f_t) + \text{Const.}$$

- Уберём константы:

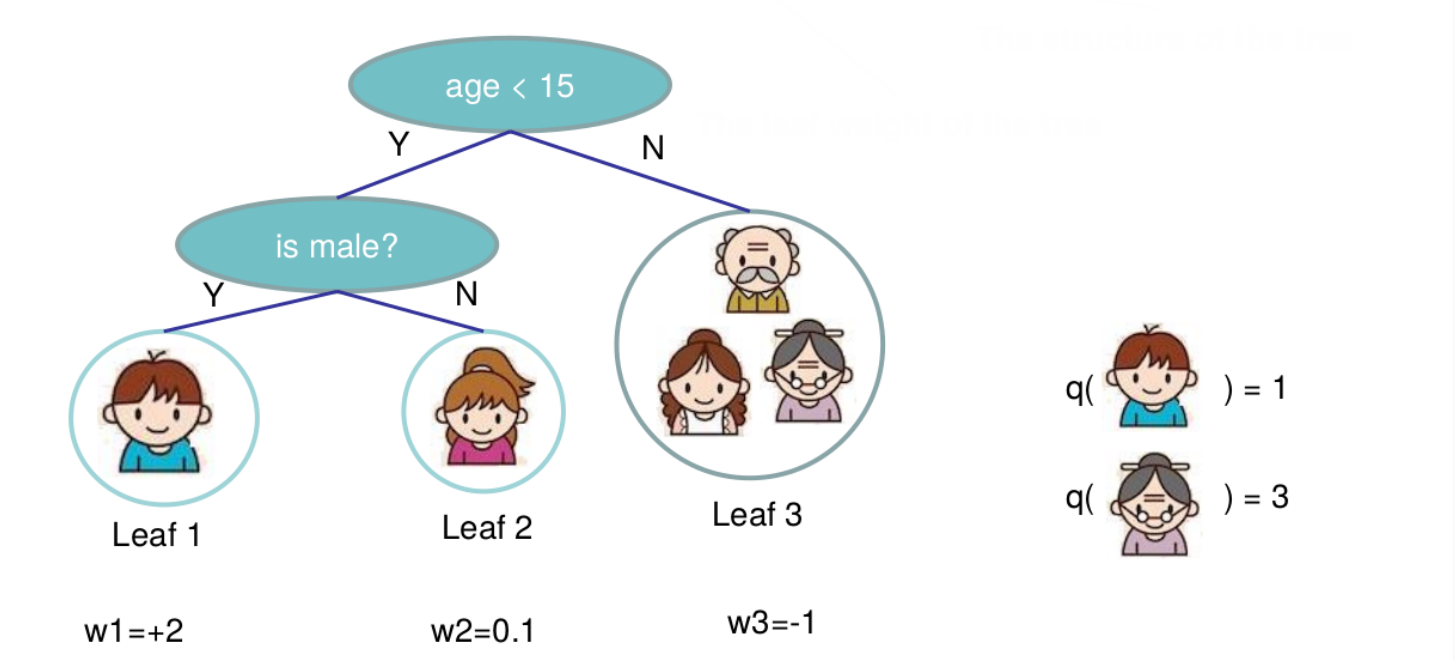
$$\text{Obj}^{(t)} \approx \sum_{i=1}^{N} \left(g_i f_t(x_i) + \frac{1}{2} h_i f_t^2(x_i) \right) + \Omega(f_t).$$

- И это и есть основная идея *градиентного бустинга*.
- Теперь давайте вернёмся к обучению деревьев и сложим всё воедино.

Градиентный бустинг

- Дерево – это вектор оценок в листьях и функция, которая вход отображает в лист:

$$f_t(x) = w_{q(x)}, \text{ где } w \in \mathbb{R}^T, \; q : \mathbb{R}^d \to \{1, \ldots, T\}.$$



- Сложность дерева можно определить как $\Omega(f_t) = \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} w_j^2$.

Градиентный бустинг

- Теперь перегруппируем слагаемые относительно листьев:

$$\begin{aligned}
\text{Obj}^{(t)} &\approx \sum_{i=1}^{N} \left(g_i f_t(x_i) + \frac{1}{2} h_i f_t^2(x_i) \right) + \Omega(f_t) \\
&= \sum_{i=1}^{N} \left(g_i w_{q(x_i)} + \frac{1}{2} h_i w_{q(x_i)}^2 \right) + \Omega(f_t) \\
&= \sum_{j=1}^{T} \left(w_j \sum_{i \in I_j} g_i + \frac{1}{2} w_j^2 \left(\sum_{i \in I_j} h_i + \lambda \right) \right) + \gamma T \\
&= \sum_{j=1}^{T} \left(G_j w_j + \frac{1}{2} w_j^2 \left(H_j + \lambda \right) \right) + \gamma T,
\end{aligned}$$

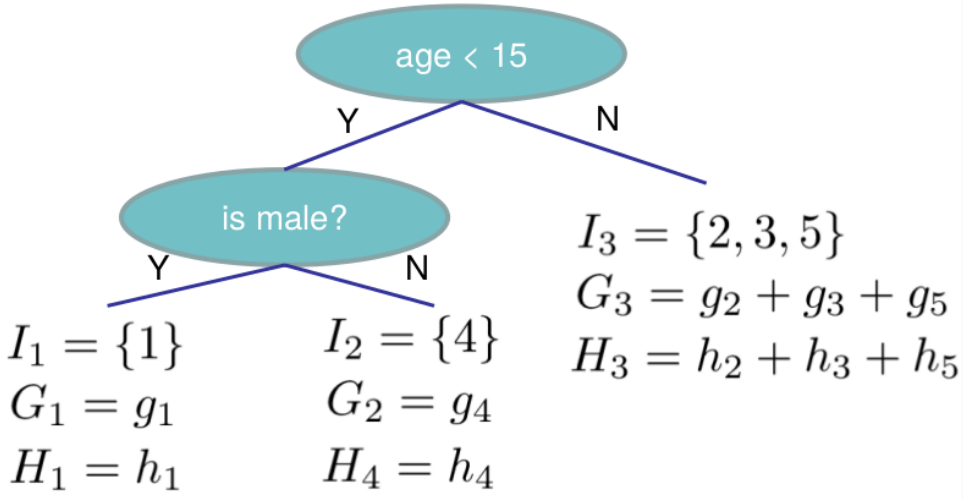
где $I_j = \{i \mid q(x_j) = i\}$, $G_i = \sum_{i \in I_j} g_i$, $H_i = \sum_{i \in I_j} h_i$.

- Это сумма T независимых квадратичных функций, так что

$$w_j^* = -\frac{G_j}{H_j + \lambda}, \quad \text{Obj}^{(t)} \approx -\frac{1}{2} \sum_{j=1}^{T} \frac{G_j^2}{H_j + \lambda} + \gamma T.$$

Градиентный бустинг

- Пример из (Chen, Guestrin):



$$Obj = -\sum_j \frac{G_j^2}{H_j + \lambda} + 3\gamma$$

The smaller the score is, the better the structure is

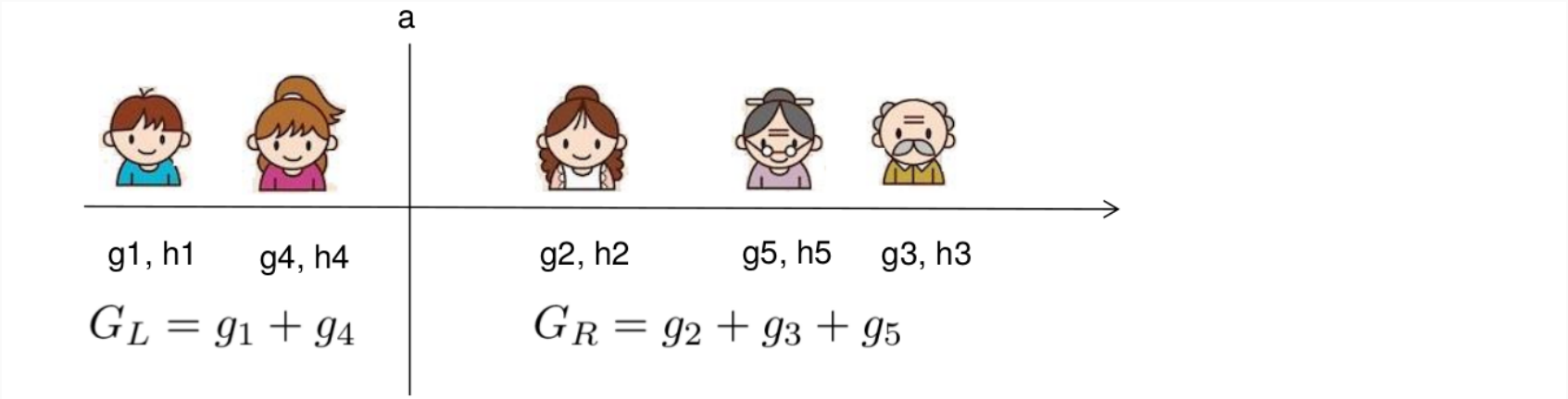
Градиентный бустинг

- Так что мы находим наилучшую структуру дерева относительно $-\frac{1}{2}\sum_{j=1}^{T}\frac{G_j^2}{H_j+\lambda}+\gamma T$ и используем оптимальные веса листьев $w_j^* = -\frac{G_j}{H_j+\lambda}$.
- Как найти структуру? Жадно: для каждого листа попробуем добавить разбиение, и целевая функция меняется на

$$\text{Gain} = \frac{1}{2}\left(\frac{G_L^2}{H_L+\lambda}+\frac{G_R^2}{H_R+\lambda}-\frac{(G_L+G_R)^2}{H_L+H_R+\lambda}\right)-\gamma.$$

Градиентный бустинг

- Самое лучшее разбиение – то, которое максимизирует gain:



- Обрезание: сначала вырастим дерево до максимальной глубины, потом рекурсивно обрежем листья с отрицательным gain.

MatrixNet

- Разработанный в «Яндекс» вариант градиентного бустинга для ранжирования – *MatrixNet*.
- Точные детали его не опубликованы, но основные особенности известны:
  - *oblivious decision trees* – все узлы одного уровня обязательно используют один и тот же атрибут; это дополнительная регуляризация, помогающая выделять меньше и более полезных признаков;
  - вместо ограничений на число сэмплов в листе – регуляризация самих значений в листьях;
  - сложность модели в бустинге зависит от итерации (сначала простые, потом более сложные).
- А основная идея та же самая.

Learning to rank

Метрики ранжирования

- Ещё одна важная постановка задачи – *learning to rank* (ранжирование).
- Задача поиска: выдать ранжированный список наиболее релевантных документов по запросу.
- Классические метрики:
  (1) точность (precision) – количество «хороших» (релевантных запросу) документов в выдаче, делённое на общее количество документов в выдаче;
  (2) полнота (recall) – количество «хороших» документов в выдаче, делённое на общее число релевантных документов в базе поисковой системы.
- Однако здесь те же проблемы; эти параметры не зависят от ранжирования выдачи, надо знать заранее, сколько потребуется рекомендаций.

Метрики ранжирования

- Метрики качества ранжирования:
  - NDCG, Normalized Discounted Cumulative Gain; выберем топ-k рекомендаций (k может быть заведомо больше нужного числа) и посчитаем:

$$\mathrm{DCG}_k = \sum_{i=1}^{k} \frac{2^{\hat{r}_i} - 1}{\log_2(1+i)},$$

$$\mathrm{NDCG}_k = \frac{\mathrm{DCG}_k}{\mathrm{IDCG}_k},$$

    где $\hat{r}_i$ – наша оценка рейтинга продукта на позиции i, а IDCG_k – значение DCG_k при ранжировании по истинным значениям (рейтингам из валидационного набора);
  - NDCG от 0 до 1, но ей трудно придумать естественную интерпретацию (как вероятность чего-нибудь, например).

Метрики ранжирования

- Метрики качества ранжирования:
  - AUC, Area Under (ROC) Curve; можно считать по всей выдаче сразу;
  - AUC – вероятность того, что случайно выбранная пара продуктов с разными оценками будет отранжирована правильно (релевантный будет выше в выдаче, чем нерелевантный);
  - в бинарном случае можно посчитать в замкнутом виде:

$$\hat{A} = \frac{S_0 - n_0(n_0+1)/2}{n_0 n_1},$$

где n_0, n_1 – число релевантных и нерелевантных запросу документов, $S_0 = \sum p_i$ – сумма номеров позиций релевантных объектов в выдаче.

Метрики ранжирования

- Метрики, основанные на каскадных моделях пользователей.
- ERR (Expected Reciprocal Rank) – ожидаемый обратный ранг документа, на котором остановится пользователь:

$$\mathrm{ERR} = \sum_{i=1}^{n} \frac{1}{i} p(\text{пользователь остановится на } i) =$$

$$= \sum_{i=1}^{n} \frac{1}{i} R(y_i) \prod_{j=1}^{i-1} (1 - R(y_i)),$$

  где остановка происходит, если случайное число от 0 до 1 меньше $R(y)$; часто используют $R(r) = \frac{2^r - 1}{2^{r_{\max}}}$.
- К сожалению, все метрики качества кусочно-постоянные. Надо что-то придумывать.

BM25

- Классическая функция BM25, ещё из 1970-80-х годов:

$$\text{score}(D, Q) = \sum_{i=1}^{n} \text{idf}(q_i) \frac{\text{tf}(q_i, D)(k_1 + 1)}{\text{tf}(q_i, D) + k_1 \left(1 - b + b \frac{|D|}{\text{Avgdoclen}}\right)},$$

  где q_i – ключевые слова из запроса Q, D – документ, k_i – параметры, которые можно обучить или выставить $k_1 \in [1.2, 2.0]$, $b = 0.75$.
- Сейчас обычно пытаются переформулировать как задачу supervised learning.
- Данные вида (Q, D, r), где r – оценка релевантности (обычно дискретная и размеченная людьми).

Три подхода

- Выделяя признаки в паре (Q, D), получим $(\mathbf{x}_j^q, r_j^q)_{q,j}$.
- Три подхода к тому, чтобы сделать непрерывную целевую функцию:
  - *pointwise* (поточечный): для функции ошибки ℓ (например, ошибка регрессии или классификации)

$$\sum_{q,j} \ell(f(\mathbf{x}_j^q), r_j^q) \to \min;$$

  - *pairwise* (попарный): правильно упорядочиваем пары с разными оценками

$$\sum_q \sum_{i,j: r_i^q > r_j^q} \ell(f(\mathbf{x}_i^q) - f(\mathbf{x}_j^q)) \to \min;$$

  - *listwise* (списочный): определим функцию потери на всём списке документов, ассоциированных с запросом,

$$\ell\left(\{f(\mathbf{x}_j^q)\}_{j=1}^{m_q}, \{r_j^q\}_{j=1}^{m_q}\right) \to \min\,.$$

Развитие learning to rank

- Обычно подходы работают на pairwise-ошибке:
  - RankSVM: берём SVM с ошибкой $\ell(t) = \max(0, 1 - t)$ и обучаем на попарных сравнениях;
  - RankBoost, RankNet, LambdaRank (поговорим о них).
- Публичные датасеты и большие продвижения около 2010:
  - «Интернет-математика» от Яндекса (2009),
  - Microsoft Learning to Rank Datasets (2010),
  - Yahoo! Learning to Rank Challenge (2010).

RankNet

RankNet

- RankNet – первая идея pairwise-подхода.
- Пусть у нас есть кое-какие прямые данные для обучения (т.е. про некоторые подмножества документов эксперт сказал, какие более релевантны, какие менее).
- Подход к решению: давайте обучать функцию, которая по данному вектору атрибутов $\mathbf{x} \in \mathbb{R}^n$ выдаёт $f(\mathbf{x})$ и ранжирует документы по значению $f(\mathbf{x})$.

RankNet

- Итак, для тестовых примеров $\mathbf{x}_i$ и $\mathbf{x}_j$ модель считает $s_i = f(\mathbf{x}_i)$ и $s_j = f(\mathbf{x}_j)$, а затем оценивает

$$p_{ij} = p(\mathbf{x}_i \succ \mathbf{x}_j) = \frac{1}{1 + e^{-\alpha(s_i - s_j)}}.$$

- А данные – это на самом деле $q(\mathbf{x}_i \succ \mathbf{x}_j)$, либо точные из $\{0, 1\}$, либо усреднённые по нескольким экспертам.
- Поэтому разумная функция ошибки – кросс-энтропия

$$C = -q_{ij} \log p_{ij} - (1 - q_{ij}) \log(1 - p_{ij}).$$

RankNet

- Ошибка: $C = -q_{ij} \log p_{ij} - (1 - q_{ij}) \log(1 - p_{ij})$.
- Для самого частого случая, когда оценки релевантности точные, и $q_{ij} = (1 + S_{ij})/2$ для $S_{ij} \in \{-1, 0, +1\}$, мы получаем

$$C = \frac{1}{2}(1 - S_{ij})\alpha(s_i - s_j) + \log\left(1 + e^{-\alpha(s_i - s_j)}\right), \text{ т.е.}$$

$$C = \begin{cases} \log\left(1 + e^{-\alpha(s_i - s_j)}\right), & \text{если } S_{ij} = 1, \\ \log\left(1 + e^{-\alpha(s_j - s_i)}\right), & \text{если } S_{ij} = -1. \end{cases}$$

- Т.е. ошибка симметрична, что уже добрый знак.

RankNet

- Ошибка: $C = -q_{ij} \log p_{ij} - (1 - q_{ij}) \log(1 - p_{ij})$.
- Давайте подсчитаем градиент по s_i:

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

- И теперь осталось использовать этот подсчёт для градиента по весам:

$$\frac{\partial C}{\partial w_k} = \sum_i \frac{\partial C}{\partial s_i} \frac{\partial s_i}{\partial w_k} + \sum_j \frac{\partial C}{\partial s_j} \frac{\partial s_j}{\partial w_k}.$$

RankNet

- Основной пафос RankNet – в том, что это можно факторизовать:

$$\frac{\partial C}{\partial w_k} = \sum_i \frac{\partial C}{\partial s_i}\frac{\partial s_i}{\partial w_k} + \sum_j \frac{\partial C}{\partial s_j}\frac{\partial s_j}{\partial w_k} = \lambda_{ij}\left(\frac{\partial s_i}{\partial w_k} - \frac{\partial s_j}{\partial w_k}\right),$$

где

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

- Переупорядочив пары так, чтобы всегда было $\mathbf{x}_i \succ \mathbf{x}_j$ и $S_{ij} = 1$, получим

$$\lambda_{ij} = \frac{\partial C(s_i - s_j)}{\partial s_i} = -\alpha\frac{1}{1 + e^{\alpha(s_i - s_j)}}.$$

RankNet

- $\lambda_{ij} = \frac{\partial C(s_i - s_j)}{\partial s_i} = -\alpha \frac{1}{1+e^{\alpha(s_i - s_j)}}$.
- Значит, если для данной выдачи есть множество пар I, в которых известно, что $\mathbf{x}_i \succ \mathbf{x}_j$, $(i,j) \in I$, то суммарный апдейт для веса w_k будет

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

$$\text{где } \lambda_i = \sum_{j:(i,j)\in I} \lambda_{ij} - \sum_{j:(j,i)\in I} \lambda_{ij}.$$

- И можно просто считать λ_i по таким mini-batches от каждого запроса, а потом уже апдейтить.
- Иначе говоря, λ_i «тянет» ссылку в выдаче вверх или вниз, и мы апдейтим веса на основе этого.

LambdaRank

LambdaRank

- Проблема с RankNet в том, что оптимизируется число попарных ошибок, а это не всегда то, что нужно.
- Градиенты RankNet – это не то же самое, что градиенты NDCG:

- Как оптимизировать, скажем, NDCG?

LambdaRank

- Заметим, что нам сама ошибка не нужна, а нужны только градиенты λ (стрелочки).
- Давайте просто представим себе мифическую функцию ошибки C, у которой градиент

$$\lambda_{ij} = \frac{\partial C(s_i - s_j)}{\partial s_i} = \frac{-\alpha}{1 + e^{\alpha(s_i - s_j)}} \left|\Delta_{\mathrm{NDCG}}\right|,$$

  где Δ_{NDCG} – это то, на сколько NDCG изменится, если поменять i и j местами.
- То есть мы считаем градиенты уже после сортировки документов по оценкам, и градиенты как будто от NDCG.

LambdaRank

- NDCG нужно максимизировать, так что берём

$$\Delta w_k = \eta \frac{\partial C}{\partial w_k}, \text{ и тогда}$$

$$\delta C = \frac{\partial C}{\partial w_k} \delta w_k = \eta \left(\frac{\partial C}{\partial w_k} \right)^2 > 0.$$

- Оказывается, что такой подход фактически напрямую оптимизирует NDCG (сглаженную версию).
- Мощная идея: можно не знать функцию, а просто придумать разумные градиенты; чтобы под них существовала функция, в разумных случаях достаточно (лемма Пуанкаре), чтобы сходились вторые частные производные.

RankBoost

RankBoost

- RankBoost: задачу ранжирования по массе разных фич, характеризующих документы и запросы, можно решить и бустингом.
- Формально говоря, нам надо обучить функцию $F(\mathbf{x}) : \mathbb{R}^n \to \mathbb{R}$ по входу $\{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$ и функции частичных предпочтений $\Phi : \mathbf{X} \times \mathbf{X} \to \mathbb{R}$ ($\Phi(\mathbf{x}_i, \mathbf{x}_j) > 0$, если $\mathbf{x}_i$ лучше $\mathbf{x}_j$, и т.д.).
- Обычно в качестве функции Φ подаётся просто разбиение на «хорошие» (например, релевантные) и «плохие»: $\Phi(\mathbf{x}, \mathbf{y}) = 1$ для хорошего $\mathbf{x}$ и плохого $\mathbf{y}$ и 0, если они из одного множества.

RankBoost

- RankBoost по сути работает примерно как AdaBoost, но раньше веса давали распределение на примерах, а теперь – на парах примеров: инициализируем распределение $D^{(1)} = D(\mathsf{x}, \mathsf{y})$ на $\mathsf{X} \times \mathsf{X}$, а потом для $m = 1..M$:
  1. обучаем слабую ранжирующую функцию $h_m(\mathsf{x}) : \mathsf{X} \to \mathbb{R}$ по распределению $D^{(m)}$;
  2. выбираем $\alpha_m \in \mathbb{R}$ (потом скажу как);
  3. пересчитываем новое распределение

$$D^{(m+1)}(\mathsf{x}, \mathsf{y}) = \frac{1}{Z_m} D^{(m)}(\mathsf{x}, \mathsf{y}) e^{\alpha_m (h_m(\mathsf{x}) - h_m(\mathsf{y}))}.$$

  4. После обучения ранжируем как $H_M(\mathsf{x}) = \sum_{m=1}^{M} \alpha_m h_m(\mathsf{x})$.

RankBoost

- Тогда получится такая теорема: если вернуться от $D^{(m+1)}(x, y)$ к $D(x, y)$, будет

$$D^{(m+1)}(x, y) = \frac{1}{\prod_m Z_m} D(x, y) e^{H_M(x) - H_M(y)}.$$

- Значит, ошибку можно оценить как

$$J_M = \sum_{x,y} D(x, y) \left[H_M(x) \geq H_M(y)\right] \leq$$

$$\leq \sum_{x,y} D(x, y) e^{H_M(x) - H_M(y)} = \sum_{x,y} D^{(m+1)}(x, y) \prod_m Z_m = \prod_m Z_m.$$

RankBoost

- И выбирать α_m можно (и нужно) так, чтобы минимизировать $\prod_m Z_m$, т.е. на шаге m минимизировать

$$Z_m = \sum_{x,y} D^{(m)}(x, y) e^{\alpha_m (h_m(x) - h_m(y))}.$$

- Формально для нас h_m – чёрный ящик, но на практике мы часто выбираем алгоритм обучения и для h_m, так что его тоже можно выбирать так, чтобы минимизировать Z_m.

RankBoost

- Теорема: для любого слабого ранжирования h $Z(\alpha)$ имеет единственный минимум, так что можно просто бинарным поиском.
- Если $h \in \{0, 1\}$, можно и аналитически: обозначим $W_b = \sum_{\mathsf{x},\mathsf{y}} D(\mathsf{x}, \mathsf{y})\,[h(\mathsf{x}) - h(\mathsf{y}) = b]$. Тогда

$$\alpha_{\text{opt}} = \frac{1}{2} \ln \left(\frac{W_{-1}}{W_{+1}} \right), \quad Z = W_0 + 2\sqrt{W_{-1} W_{+1}}.$$

RankBoost

- А для $h \in [0, 1]$ можно приблизить: $e^{\alpha x} \leq \left(\frac{1+x}{2}\right) e^{\alpha} + \left(\frac{1-x}{2}\right) e^{-\alpha}$, $x \in [-1, 1]$, так что

$$Z \leq \left(\frac{1-r}{2}\right) e^{\alpha} + \left(\frac{1+r}{2}\right) e^{-\alpha}, \quad r = \sum_{\mathsf{x},\mathsf{y}} D(\mathsf{x}, \mathsf{y}) \left(h(\mathsf{x}) - h(\mathsf{y})\right),$$

  и можно выбирать

$$\alpha_{\mathrm{opt}} = \frac{1}{2} \ln \left(\frac{1+r}{1-r}\right),$$

  а h можно обучать так, чтобы максимизировать $|r|$.

- Но можно и ещё лучше...

MART

Деревья регрессии

- Теперь давайте градиентный бустинг применять к задаче ранжирования более напрямую.
- Начнём с чуть изменённых обозначений для деревьев принятия решений в случае регрессии.
- Рассмотрим датасет $\mathsf{X} = \{\mathbf{x}_i, y_i\}$.
- Можно определить *regression stump* (регрессионный пень) так: выбираем одну координату (атрибут) j (т.е. берём x_{ij}) и ищем там оптимальное разбиение – такое значение порога t, что

$$S_j = \sum_{i \in \text{Left}} (y_i - \mu_{\text{Left}})^2 + \sum_{i \in \text{Right}} (y_i - \mu_{\text{Right}})^2$$

минимизируется, где Left и Right – множества точек слева и справа от t по j-й координате.

Деревья регрессии

- Если повторить эту процедуру L раз (как-то выбирая для расщепления листья – например, по максимальной дисперсии), получится регрессионное дерево с L листьями.
- В каждом листе определим γ_l – среднее по y_i из этого листа.
- Тогда, чтобы применить регрессионное дерево, надо по нему спуститься до листа и взять γ_l из этого листа.

MART

- MART – это бустинг, сделанный на регрессионных деревьях.
- Иначе говоря, окончательная модель будет, опять же, по $\mathbf{x} \in \mathbb{R}^d$ искать $y \in \mathbb{R}$, и искать в виде

$$F_M(\mathbf{x}) = \sum_{m=1}^{M} \alpha_m f_m(\mathbf{x}),$$

где $f_m(\mathbf{x})$ задаётся регрессионным деревом, а $\alpha_m \in \mathbb{R}$ – веса бустинга, и в процессе обучения обучаются одновременно f_m и α_m.

MART

- Нам нужно понять, как обучать новое дерево F_{m+1}, если мы уже обучили m деревьев.
- Зафиксируем функцию ошибки C (она дана свыше).
- Идея: следующее дерево моделирует производные ошибки по текущей модели в точках из датасета:

$$\Delta C \approx \frac{\partial C(F_m)}{\partial F_m} \Delta F,$$

и ΔC будет отрицательным, если взять для $\eta > 0$

$$\Delta F = -\eta \frac{\partial C(F_m)}{\partial F_m}.$$

- Непараметрический метод: мы рассматриваем F_m *в каждой точке из датасета* как «параметр» и по ним оптимизируем.

MART

- Пример: бинарная классификация, $y_i \in \{\pm 1\}$, $\mathbf{x} \in \mathbb{R}^n$, $F(\mathbf{x})$.
- Обозначим $p_+ = p(y = 1 \mid \mathbf{x})$, $p_- = p(y = -1 \mid \mathbf{x})$, $I_+(\mathbf{x}_i) = [y_i = 1]$, $I_-(\mathbf{x}_i) = [y_i = -1]$.
- Будем использовать (как и в RankNet, кстати) перекрёстную энтропию

$$L(y, F) = -I_+ \log p_+ - I_i \log p_i.$$

MART

- Логистическая регрессия моделирует log odds:

$$F_N(\mathbf{x}) = \frac{1}{2} \log \left(\frac{p_+}{p_-} \right),$$

$$p_+ = \frac{1}{1 + e^{-2\alpha F_m(\mathbf{x})}}, \quad p_- = \frac{1}{1 + e^{2\alpha F_m(\mathbf{x})}},$$

и мы получаем

$$L(y, F) = \log \left(1 + e^{-2y\alpha F_m(\mathbf{x})} \right).$$

MART

- $L(y, F) = \log\left(1 + e^{-2y\alpha F_m(\mathbf{x})}\right)$.
- От этой функции легко взять производную по $F(\mathbf{x})$:

$$\bar{y}_i = \left[\frac{\partial L(y_i, F(\mathbf{x}_i))}{\partial F(\mathbf{x}_i)}\right]_{F(\mathbf{x})=F_m(\mathbf{x})} = \frac{2y_i\alpha}{1 + e^{2y_i\alpha F_m(\mathbf{x})}}.$$

- И мы строим новое регрессионное дерево, которое пытается смоделировать $\bar{y}_i$.

MART

- Осталось только выбрать вес, с которым это новое дерево войдёт в сумму.
- Мы хотим выбрать (примерно) оптимальный шаг для каждого листа, т.е. минимизировать потери:

$$\gamma_{lm} = \arg\min_{\gamma} \log\left(1 + e^{2y_i\alpha(F_m(\mathbf{x})+\gamma)}\right) = \arg\min_{\gamma} g(\gamma).$$

- Минимизировать можно итеративно по методу Ньютона-Рапсона: $\gamma_{n+1} = \gamma_n - \frac{g'(\gamma_n)}{g''(\gamma_n)}$.

MART

- И мы аппроксимируем одним шагом этого метода, начиная с нуля:

$$\gamma_{lm} = -\frac{g'}{g''} = \frac{\sum_{x_i \in R_{lm}} \bar{y}_i}{\sum_{x_i \in R_{lm}} |\bar{y}_i| \left(2\alpha - |\bar{y}_i|\right)}.$$

Упражнение. Проверьте эту формулу.

LambdaMART

LambdaMART

- Теперь уже понятно, как из MART сделать LambdaMART.
- Мы просто добавим в градиенты целевую метрику, например

$$\lambda_{ij} = S_{ij} \left| \Delta \mathrm{NDCG} \frac{\partial C_{ij}}{\partial o_{ij}} \right|, \quad o_{ij} = F(x_i) - F(x_j).$$

- Функция ошибки нам тоже уже известна:

$$\begin{aligned} C_{ij} =& C(o_{ij}) = s_j - s_i + \log\left(1 + e^{s_i - s_j}\right), \\ \frac{\partial C_{ij}}{\partial o_{ij}} =& \frac{\partial C_{ij}}{\partial s_i} = -\frac{1}{1 + e^{o_{ij}}}. \end{aligned}$$

LambdaMART

- Получается, что знак λ_{ij} зависит только от меток i и j, и в каждой точке мы можем собрать все «действующие силы» как
$$\lambda_i = \sum_j \lambda_{ij}.$$
- Это и называется LambdaMART.
- Вариант: LambdaSMART (submodel): мы инициализируем первое дерево какой-нибудь обученной хорошей базовой моделью, а всё дальнейшее – это её уточнение.

LambdaMART

- Остался только один вопрос: откуда взять веса α_i (при $f_i(\mathbf{x})$)?
- Базовый LambdaMART выбирает эти веса индивидуально для каждого листа дерева.
- Мы хотим сдвинуться в сторону минимума ошибки; значит, надо идти в сторону нуля производной. Один шаг метода Ньютона-Рапсона:

$$F_m(x_i) = F_{m-1}(x_i) + v \sum_l \gamma_{lm} \left[x_i \in R_{lm}\right],$$

где $\gamma_{lm} = \dfrac{F'_m(x_i)}{F''_m(x_i)}$, а v – регуляризатор.

LambdaMART

- Первая производная – это просто λ_i; вторая производная – λ_i':

$$\gamma_{lm} = \frac{\sum_{x_i \in R_{lm}} \lambda_i}{\sum_{x_i \in R_{lm}} \frac{\partial \lambda_i}{\partial F_m(x_i)}}$$

- И теперь можно собрать весь алгоритм целиком.

LambdaSMART: алгоритм

1. $F_0(x_i) = \mathrm{BaseModel}(x_i)$, $i = 0..N$.
2. Для m от 1 до M (числа деревьев в сумме):
   - 2.1 $y_i = \lambda_i = \sum_j \lambda_{ij}$, $i = 0..N$ (считаем градиенты);
   - 2.2 $w_i = \frac{\partial y_i}{\partial F(x_i)}$, $i = 0..N$ (вторые производные);
   - 2.3 строим новое дерево $\{R_{lm}\}_{l=1}^{L}$ на вершинах $\{y_i, x_i\}_{n=1}^{N}$;
   - 2.4 $\gamma_{lm} = \frac{\sum_{x_i \in R_{lm}} y_i}{\sum_{x_i \in R_{lm}} w_i}$, $l = 1..L$ (веса узлов);
   - 2.5 $F_m(x_i) = F_{m-1}(x_i) + v \sum_l \gamma_{lm} \left[x_i \in R_{lm}\right]$, $i = 0..N$.

Как оптимизировать α_m

- В Lambda[S]MART веса бустинга подбираются как шаг метода Ньютона для каждого листа дерева.
- Но благодаря тому, что наши IR-метрики дискретные, можно просто подобрать оптимальный α_m.
- Давайте рассмотрим общую задачу: предположим, что у нас есть две ранжирующих функции, R и R', и мы хотим их оптимально линейно скомбинировать, т.е. подобрать оптимальный коэффициент α в

$$s_i = (1 - \alpha)s_i^R + \alpha s_i^{R'}.$$

Как оптимизировать α_m

- Идея простая: представьте себе, как α меняется от 0 (в чистом виде R) до 1 (в чистом виде R').
- Тогда метрики вроде NDCG реально меняются только в точках пересечения (да и то не всегда, а только если пересеклись документы с разными метками).
- Ну вот и давайте просто переберём все такие пары, найдём их точки пересечения и составим список интересных значений α.
- А затем отсортируем список, подсчитаем метрику в каждом интервале и найдём оптимальный α; для ситуации бустинга просто надо это делать на каждом шаге.
- Кажется, что это очень тупо, но на самом деле это квадратичный алгоритм, что не так уж страшно.

Как оптимизировать α_m

- LambdaMART – победитель в Yahoo! Learning to Rank Challenge (2011).
- Что было нового с тех пор?
- Plackett-Luce model выражает ранжирование в виде вероятностей:
  - выбираем первый документ с вероятностью, пропорциональной релевантности;
  - выбираем второй документ из остальных тоже пропорционально релевантности...
- Это даёт непрерывную listwise-ошибку, которую можно оптимизировать.
- BayesRank оптимизирует её, MatrixNet, видимо, тоже.

Спасибо!

Спасибо за внимание!