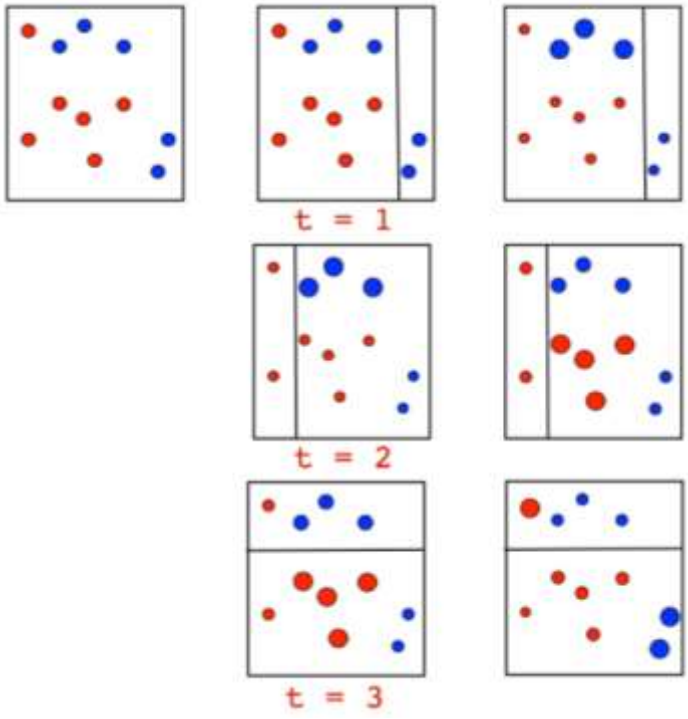
ЛЕКЦИЯ 16.1 Градиентный бустинг

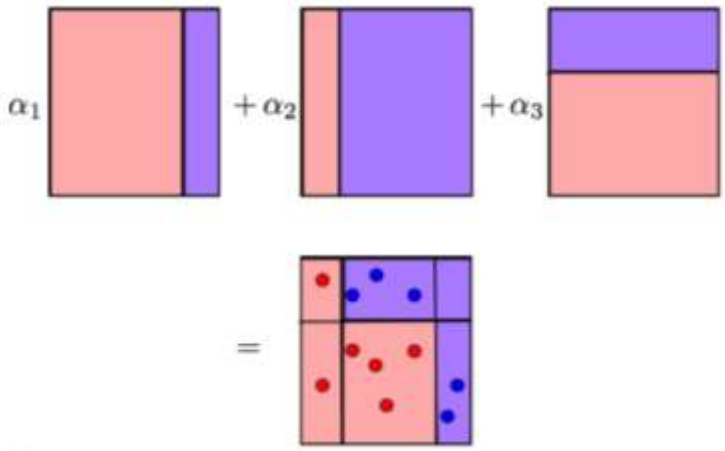
Как мы поняли бустинг, в свою очередь, воплощает идею последовательного построения линейной комбинации алгоритмов. Каждый следующий алгоритм старается уменьшить ошибку текущего ансамбля. Бустинг родился из вопроса: можно ли из большого количества относительно слабых и простых моделей получить одну сильную модель? Говоря «слабые модели», мы имеем в виду не простые базовые модели, такие как деревья решений, а модели с низкими показателями точности, где плохое немного лучше, чем случайное.

Положительный математический ответ на этот вопрос был найден, но потребовалось несколько лет, чтобы разработать полноценные алгоритмы на основе этого решения, например, AdaBoost (от адаптивность и усиление). Эти алгоритмы используют жадный подход: сначала они строят линейную комбинацию простых моделей (базовые алгоритмы) путем повторного взвешивания входных данных. Затем модель (обычно дерево решений) строится на ранее неверно предсказанных объектах, которым теперь присваиваются большие веса. Поскольку AdaBoost объединился с GBM (Gradient Boosting Machine – Машина усиления градиента), стало очевидно, что AdaBoost — это всего лишь конкретная разновидность GBM.

Сам алгоритм имеет очень четкую визуальную интерпретацию и интуитивно понятное определение весов. Давайте посмотрим на следующую задачу классификации игрушек, где мы собираемся разделить данные между деревьями глубины 1 (также известными как «пни (от слова пень)») на каждой итерации AdaBoost. Для первых двух итераций имеем следующую картину:



Размер точки соответствует ее весу, присвоенному за неверный прогноз. На каждой итерации видно, что веса растут — пни не справляются с этой проблемой. Хотя, если мы возьмем взвешенное голосование за пни, мы получим правильные классификации:



Псевдокод:

- Инициализировать выборочные веса $w_i^{(0)} = \frac{1}{l}, i = 1, \dots, l.$
- Для всех $t = 1, \dots, T$
  - Алгоритм тренировочной базы b_t, позволяет определить ошибку обучения ϵ_t
  - $\propto_t = \frac{1}{2} \ln \frac{1-\epsilon_t}{\epsilon_t}$
  - Обновить веса выборки: $w_i^{(t)} = w_i^{(t-1)} e^{-\propto_t y_i b_t(x_i)}, i = 1, \dots, l$
  - Нормировать веса выборки: $w_0^{(t)} = \sum_{j=1}^{l} w_j^{(t)}, w_i^{(t)} = \frac{w_i^{(t)}}{w_0^{(t)}}, i = 1, \dots, l$
- Возвращаться $\sum_t^T \propto_t b_t$

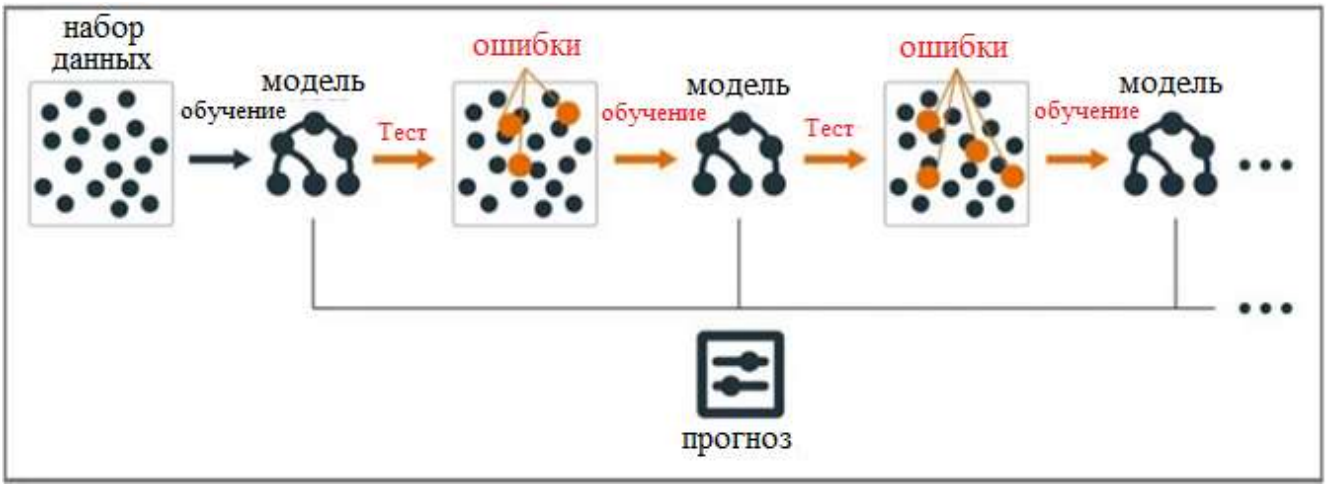
AdaBoost работает хорошо. Однако существует проблема переобучения, особенно когда данные имели сильные выбросы. Поэтому в подобных задачах AdaBoost работал нестабильно.

В 1999 году Джером Фридман придумал обобщение разработки алгоритмов повышения — Gradient Boosting (Machine), также известное как GBM. Фридман заложил статистическую основу для многих алгоритмов, обеспечивающих общий подход повышения для оптимизации в функциональном пространстве. Многие реализации GBM также появились под разными именами и на разных платформах: Stochastic GBM, GBDT (Gradient Boosted Decision Trees – деревья решений с градиентным усилением), GBRT (Gradient Boosted Regression Trees – деревья регрессии с градиентным усилением), MART (Multiple Additive Regression Trees – деревья множественной аддитивной регрессии) и другие. Кроме того, сообщество машинного обучения было очень сегментированным и разрозненным,

что затрудняло отслеживание того, насколько широкое распространение получил данный подход.

Интуиция

Gradient Boosting опирается на интуицию, согласно которой наилучшая возможная следующая модель в сочетании с предыдущими моделями сводит к минимуму общие ошибки прогнозирования. Основная идея состоит в том, чтобы установить целевые результаты из предыдущих моделей в следующую модель, чтобы свести к минимуму ошибки. Это еще один алгоритм повышения (несколько других — Adaboost, XGBoost и т. д.).



Рассмотрим задачу регрессии с квадратичной функцией потерь:

$$\mathcal{L}(y,x) = C\sum_{i=1}^{N}\big(y_i - a(x_i)\big)^2 \to min$$

для некоторой константы C; значение константы не влияет на решение и может быть проигнорировано, если установить его равным 1.

Для решения будем строить композицию из K базовых алгоритмов

$$a(x) = a_K(x) = b_1(x) + b_2(x) + \cdots + b_K(x)$$

Если мы обучим единственное решающее дерево, то качество такой модели, скорее всего, будет низким. Однако о построенном дереве мы знаем, на каких объектах оно давало точные предсказания, а на каких ошибалось.

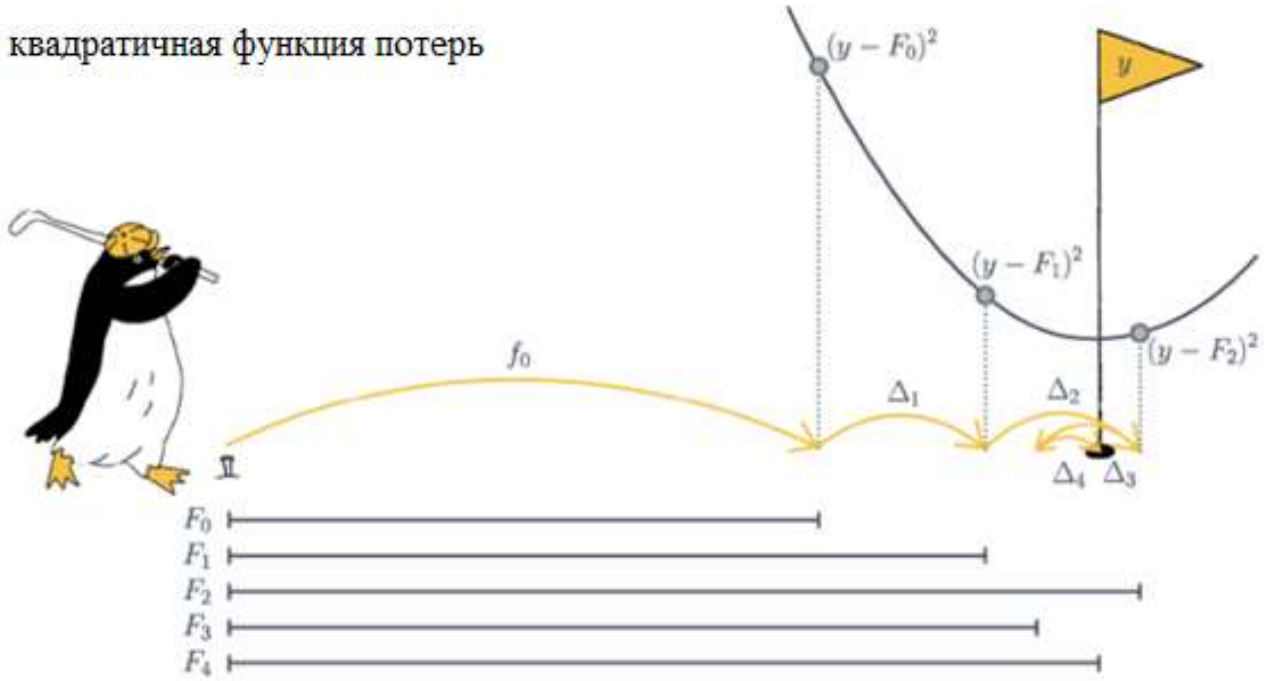
Попробуем использовать эту информацию и обучим еще одну модель. Допустим, что предсказание первой модели на объекте x_l на 10 больше, чем необходимо (т.е. $b_1(x_l) = y_l + 10$). Если бы мы могли обучить новую модель, которая на x_l будет выдавать ответ -10, то сумма ответов этих двух моделей на объекте x_l в точности совпала бы с истинным значением:

$$b_1(x_l) + b_2(x_l) = (y_l + 10) + (-10) = y_l$$

Другими словами, если вторая модель научится предсказывать разницу между реальным значением и ответом первой, то это позволит уменьшить ошибку

композиции. В реальности вторая модель тоже не сможет обучиться идеально, поэтому обучим третью, которая будет «компенсировать» неточности первых двух. Будем продолжать так, пока не построим композицию из K алгоритмов.

Для объяснения метода градиентного бустинга полезно воспользоваться следующей аналогией. Бустинг можно представить как гольфиста, цель которого – загнать мяч в лунку с координатой y_{ball}. Положение мяча здесь – ответ композиции $a(x_{ball})$. Гольфист мог бы один раз ударить по мячу, не попасть в лунку и пойти домой, но настырность заставляет его продолжить.



По счастью, ему не нужно начинать каждый раз с начальной позиции. Следующий удар гольфиста переводит мяч из текущего положения $a_k(x_{ball})$ в положение $a_{k+1}(x_{ball})$. Каждый следующий удар – это та поправка, которую вносит очередной базовый алгоритм в композицию. Если гольфист все делает правильно, то функция потерь будет уменьшаться:

$$\mathcal{L}(y, a_{k+1}(x)) < \mathcal{L}(y, a_k(x))$$

Подобно тому, как гольфист постепенно подводит мяч к цели, бустинг с каждым новым базовым алгоритмом всё больше приближает предсказание к истинному значению метки объекта.

Рассмотрим теперь другую аналогию – разложение функции в ряд Тейлора. Из курса математического анализа известно, что (достаточно хорошую) бесконечно дифференцируемую функцию $f(x)$ на интервале $x \in (a-R, a+R)$ можно представить в виде бесконечной суммы степенных функций:

$$f(x) = \sum_{n=0}^{\infty} \frac{f^{(n)}(a)}{n!}(x-a)^n$$

Одна, самая первая степенная функция в разложении, очень грубо приближает $f(x)$. Прибавляя к ней следующую, мы получим более точное приближение. Каждая следующая элементарная функция увеличивает точность приближения, но менее заметна в общей сумме. Если нам не требуется абсолютно точное разложение, вместо бесконечного ряда Тейлора мы можем ограничиться суммой его первых k элементов. Таким образом, интересующую нас функцию мы с некоторой точностью представили в виде суммы «простых» функций. Перенесём эту идею на задачи машинного обучения.

Пример с задачей регрессии: формальное описание

Рассмотрим тот же пример с задачей регрессии и квадратичной функцией потерь:

$$\mathcal{L}(y,x) = \frac{1}{2}\sum_{i=1}^{N}\big(y_i - a(x_i)\big)^2 \to min$$

Для решения также будем строить композицию из K базовых алгоритмов семейства $\mathcal{B}$:

$$a(x) = a_K(x) = b_1(x) + b_2(x) + \cdots + b_K(x)$$

В качестве базовых алгоритмов выберем семейство $\mathcal{B}$ решающих деревьев некоторой фиксированной глубины.

Используя известные нам методы построения решающих деревьев, обучим алгоритм $b_1(x) \in \mathcal{B}$, который наилучшим образом приближает целевую переменную:

$$b_1(x) = arg\min_{b\in\mathcal{B}} L(y, b(x))$$

Построенный алгоритм $b_1(x)$, скорее всего, работает не идеально. Более того, если базовый алгоритм работает слишком хорошо на обучающей выборке, то высока вероятность переобучения (низкий уровень смещения, но высокий уровень разброса). Далее вычислим, насколько сильно отличаются предсказания этого дерева от истинных значений:

$$s_i^1 = y_i - b_1(x)$$

Теперь мы хотим скорректировать $b_1(x)$ с помощью $b_2(x)$; в идеале так, чтобы $b_2(x)$ идеально предсказывал величины s_i^1, ведь в этом случае

$$a_2(x_i) = b_1(x_i) + b_2(x_i) = b_1(x_i) + s_i^1 = b_1(x_i) + \big(y_i - b_1(x)\big) = y_i$$

Найти совершенный алгоритм, скорее всего, не получится, но по крайней мере мы можем выбрать из семейства наилучшего представителя для такой

задачи. Итак, второе решающее дерево будет обучаться предсказывать разности s_i^1:

$$b_2(x_i) = arg\min_{b \in \mathcal{B}} L(s^1, b(x))$$

Ожидается, что композиция из двух таких моделей $a_2(x_i) = b_1(x_i) + b_2(x_i)$ станет более качественно предсказывать целевую переменную y.

Далее рассуждения повторяются до построения всей композиции. На k-ом шаге вычисляется разность между правильным ответом и текущим предсказанием композиции из $k-1$ алгоритмов:

$$s_i^{k-1} = y_i - \sum_{i=1}^{k-1} b_{k-1}(x_i) = y_i - a_{k-1}(x_i)$$

Затем k-й алгоритм учится предсказывать эту разность:

$$b_k(x_i) = arg\min_{b \in \mathcal{B}} \mathcal{L}(s^{k-1}, b(x))$$

а композиция в целом обновляется по формуле

$$a_k(x_i) = a_{k-1}(x_i) + b_k(x)$$

Обучение K базовых алгоритмов завершает построение композиции.

Таким образом можно выделить входные требования для Gradient Boosting:

1. Функция потерь. Например, потери L1 или средние абсолютные ошибки (MAE); потери L2 или среднеквадратическая ошибка (MSE)

2. Слабые ученики — это модели, которые используются последовательно, чтобы уменьшить ошибку, порожденную предыдущими моделями, и в конце вернуть сильную модель.

3. Аддитивная модель. При повышении градиента деревья решений добавляются по одному (последовательно), а существующие деревья в модели не изменяются.

Обобщение на другие функции потерь

Отметим теперь важное свойство функции потерь в рассмотренном выше примере с регрессией. Для этого посчитаем производную функции потерь по предсказанию $z = a_k(x_i)$ модели для i-го объекта:

$$\frac{\partial \mathcal{L}(y_i, z)}{\partial z}\Big|_{z=a_k(x_i)} = \frac{\partial}{\partial z}\frac{1}{2}(y_i - z)^2\Big|_{z=a_k(x_i)} = a_k(x_i) - y_i$$

Видим, что разность, на которую обучается k-й алгоритм, выражается через производную:

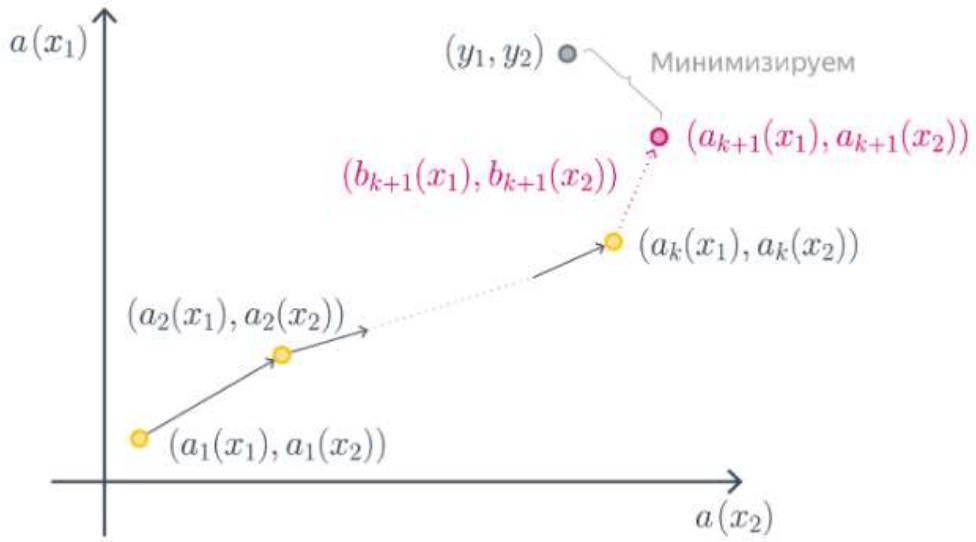
$$s_i^k = y_i - a_k(x_i) = -\frac{\partial \mathcal{L}(y_i, z)}{\partial z}\Big|_{z=a_k(x_i)}$$

Таким образом, для каждого объекта x_i очередной алгоритм в бустинге обучается предсказывать антиградиент функции потерь по предсказанию модели $-\frac{\partial \mathcal{L}(y_i,z)}{\partial z}$ в точке $a_k(x_i)$ предсказания текущей части композиции на объекте x_i.

Это наблюдение позволяет обобщить подход построения бустинга на произвольную дифференцируемую функцию потерь. Для этого мы заменяем обучение на разность s_i^k обучением на антиградиент функции потерь $(-g_i^k)$, где

$$g_i^k = \frac{\partial \mathcal{L}(y_i, z)}{\partial z}\Big|_{z=a_k(x_i)}$$

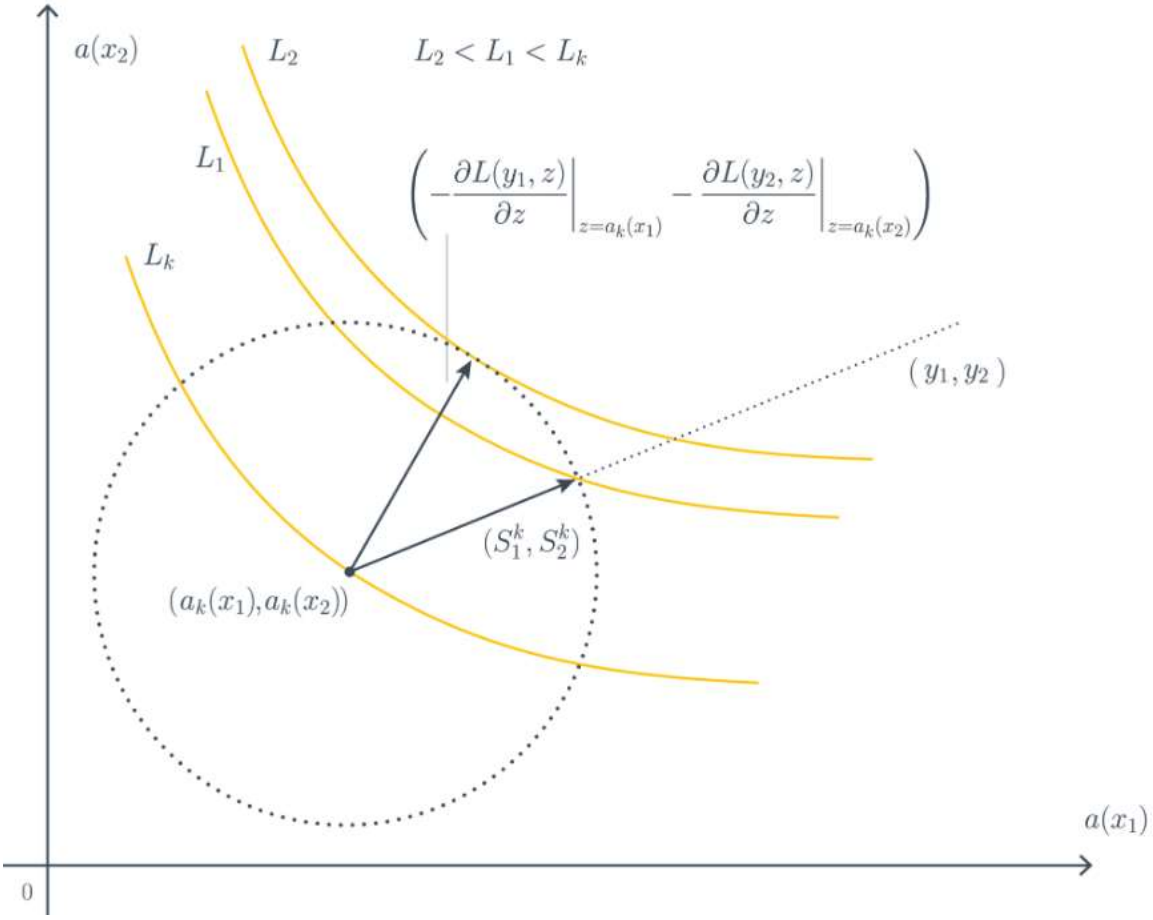
Обучение композиции можно представить (аналогиz с гольфистом) как перемещение предсказания из точки $(a_k(x_1), a_k(x_2), a_k(x_3), \dots, a_k(x_N))$ в точку $(a_{k+1}(x_1), a_{k+1}(x_2), a_{k+1}(x_3), \dots, a_{k+1}(x_N))$. В конечном итоге мы ожидаем, что точка $(a_K(x_1), a_K(x_2), a_K(x_3), \dots, a_K(x_N))$ будет располагаться как можно ближе к точке с истинными значениями $(y_1, y_2, \dots, y_K)$.



В случае квадратичной функции потерь интуиция вполне подкрепляется математикой. Изменится ли что-либо в наших действиях, если мы поменяем квадратичную функцию потерь на любую другую? С одной стороны, мы, как и прежде, можем двигаться в направлении уменьшения разности предсказания и истинного значения: любая функция потерь поощряет такие шаги для каждого отдельного объекта, ведь для любой адекватной функции потерь выполнено $\mathcal{L}(y, y) = 0$.

Но мы можем посмотреть на задачу и с другой стороны: не с точки зрения уменьшения расстояния между вектором предсказаний и вектором истинных значений, а с точки зрения уменьшения значения функции потерь. Для наискорейшего уменьшения функции потерь нам необходимо шагнуть в сторону её антиградиента по вектору предсказаний текущей композиции, то есть как раз

таки в сторону вектора $\left(-g_1^k, \ldots, -g_N^k\right)$. Это направление не обязано совпадать с шагом по направлению уменьшения разности предсказания и истинного значения. Например, может возникнуть гипотетическая ситуация, как на рисунке ниже:



В изображённом примере рассматриваются два объекта x_1 и x_2. Текущее предсказание для них – $(a_k(x_1), a_k(x_2))$, а окружность определяет варианты следующего шага: первый вариант – пойти в направлении $\left(s_1^k, s_2^k\right)$, как делалось ранее; второй – пойти в направлении антиградиента. Также показаны линии уровня значений функции потерь. Функция потерь в этом примере устроена таким образом, что $L_2 < L_1$, из-за чего шаг по антиградиенту оказывается более выгодным.

Кроме того, что движение в сторону антиградиента более выгодно с точки зрения минимизации функции потерь, это также позволяет справляться с ситуациями, когда явно посчитать остаток (т.е. разницу между целевым значением и предсказанием) не представляется возможным. Один из таких примеров это задача ранжирования. В задаче ранжирования объекты в датасете разбиты на группы и требуется построить модель, по предсказаниям которой можно было бы «правильно» упорядочить документы в каждой группе (обычно по убыванию предсказания модели). Что значит упорядочить «правильно»? Это значит, что полученная по предсказаниям модели перестановка объектов в группе

должна быть близка к идеальной по некоторой метрике. Как задается идеальная перестановка? Есть два способа:

Первый способ – это проставить каждому объекту число y, по которому можно отсортировать объекты для получения идеальной перестановки. Это число можно рассматривать как таргет (целевая метка) и обучать модель регрессии – в некоторых случаях это даже будет работать хорошо.

Второй способ – это задать набор пар объектов, которые обозначают их порядок относительно друг друга в идеальной перестановке. То есть пара (i, j) означает, что объект с номером i должен стоять раньше в перестановке, чем объект с номером j. Во втором способе таргетов у объектов нет, но дифференцируемая функция потерь есть – в библиотеке CatBoost она называется PairLogit (парный логит) и вычисляется по формуле:

$$PairLogit = \frac{-\sum_{p,n \in Pairs} \left(log \left(\frac{1}{1 + e^{-(a_p - a_n)}} \right) \right)}{|Pairs|}$$

где a_p и a_n – это предсказания модели на объектах p и n соответственно. Градиент такой функции потерь посчитать можно, а разницу между предсказанием и истинным значением – нет.

Математическое обоснование

Попробуем записать наши интуитивные соображения более формально. Пусть $\mathcal{L}$ – дифференцируемая функция потерь, а наш алгоритм $a(x)$ представляет собой композицию базовых алгоритмов:

$$a(x) = a_k(x) = b_1(x) + \cdots + b_k(x)$$

Построим композицию «жадного алгоритма»:

$$a_k(x) = a_{k-1}(x) + b_k(x)$$

где вновь добавляемый базовый алгоритм b_k обучается так, чтобы улучшить предсказания текущей композиции:

$$b_k = arg \min_{b \in \mathcal{B}} \sum_{i=1}^{N} \mathcal{L}\big(y_i, a_{k-1}(x_i) + b(x_i)\big)$$

Модель b_0 выбирается так, чтобы минимизировать потери на обучающей выборке:

$$b_0 = arg \min_{b \in \mathcal{B}} \sum_{i=1}^{N} \mathcal{L}\big(y_i, b(x_i)\big)$$

Для построения базовых алгоритмов на следующих шагах рассмотрим разложение Тейлора функции потерь $\mathcal{L}$ до первого члена в окрестности точки $\Big((y_i, a_{k-1}(x_i))\Big)$:

$$\mathcal{L}(y_i, a_{k-1}(x_i) + b(x_i)) \approx \mathcal{L}(y_i, a_{k-1}(x_i)) + b(x_i)\frac{\partial \mathcal{L}(y_i, z)}{\partial z}\Big|_{z=a_{k-1}(x_i)} =$$

$$= \mathcal{L}(y_i, a_{k-1}(x_i)) + b(x_i)g_i^{k-1}$$

Избавившись от постоянных членов, мы получим следующую оптимизационную задачу:

$$b_k \approx arg\min_{b\in\mathcal{B}} \sum_{i=1}^{N} b(x_i)g_i^{k-1}$$

Поскольку суммируемое выражение – это скалярное произведение двух векторов, его значение минимизируют $b(x_i)$, пропорциональные значениям – g_i^{k-1}. Поэтому на каждой итерации базовые алгоритмы b_k обучаются предсказывать значения антиградиента функции потерь по текущим предсказаниям композиции.

Итак, использованная нами интуиция шага в сторону «уменьшения остатка» удивительным образом привела к оптимальным смещениям в случае квадратичной функции потерь, но для других функций потерь это не так: для них смещение происходит в сторону антиградиента.

Получается, что в общем случае на каждой итерации базовые алгоритмы должны приближать значения антиградиента функции потерь. Однако есть частный случай, в котором в качестве таргета для базового алгоритма выгоднее использовать именно «остатки» – это касается функции потерь MAE (средняя абсолютная ошибка). Ее производная равна -1, 0 или +1. Приближая базовым алгоритмом антиградиент MAE, количество итераций до сходимости будет расти пропорционально масштабу таргета. То есть, если домножить целевое значение на 10, то потребуется в 10 раз больше итераций градиентного бустинга. Использование остатков в качестве таргета для базового алгоритма не имеет такой проблемы. Аналогичные рассуждения верны также для функции MAPE (средняя абсолютная ошибка в процентах), в которой проблема с масштабом таргета может проявляться еще сильнее.

Обучение базового алгоритма

При построении очередного базового алгоритма b_{k+1} мы решаем задачу регрессии с таргетом, равным антиградиенту функции потерь исходной задачи на предсказании $a_k = b_1 + \cdots + b_k$.

Теоретически можно воспользоваться любым методом построения регрессионного дерева. Важно выбрать оценочную функцию S, которая будет показывать, насколько текущая структура дерева хорошо приближает антиградиент. Её нужно будет использовать для построения критерия ветвления:

$$|R| \times S(R) - \left|R_{\text{прав}}\right| \times S\left(R_{\text{прав}}\right) - |R_{\text{лев}}| \times S(R_{\text{лев}}) \to max$$

где $S(R)$ – значение функции S в вершине $R, S\left(R_{\text{прав}}\right), S(R_{\text{лев}})$ – значения в левом и правом сыновьях потомке R после добавления предиката, $|, ,|$ – количество элементов, пришедших в вершину.

Например, можно использовать следующие оценочные функции:

$$L_2(g,p) = \sum_{i=1}^{N}(p_i - g_i)^2,\ Cos(g,p) = -\frac{\sum_{i=1}^{N}(p_i \times g_i)}{\sqrt{\sum_{i=1}^{N}(p_i^2)} \times \sqrt{\sum_{i=1}^{N}(g_i^2)}}$$

где p_i – предсказание дерева на объекте x_i, g_i – антиградиент, на который учится дерево, $p = \{p_i\}_{i=1}^{N}$, $g = \{g_i\}_{i=1}^{N}$. Функция L_2 представляет собой среднеквадратичную ошибку, а функция Cos определяет близость через косинусное расстояние между векторами предсказаний и антиградиентов.

В итоге обучение базового алгоритма проходит в два шага:

- по функции потерь вычисляется целевая переменная для обучения следующего базового алгоритма:

$$g_i^k = \left.\frac{\partial \mathcal{L}(y_i, z)}{\partial z}\right|_{z=a_k(x_i)}$$

- строится регрессионное дерево на обучающей выборке $\left(x_i, -g_i^k\right)$, минимизирующее выбранную оценочную функцию.

Поскольку для построения градиентного бустинга достаточно уметь считать градиент функции потерь по предсказаниям, с его помощью можно решать широкий спектр задач. В библиотеках градиентного бустинга даже реализована возможность создавать свои функции потерь: для этого достаточно уметь вычислять ее градиент, зная истинные значения и текущие предсказания для элементов обучающей выборки.

Типичный градиентный бустинг имеет в составе несколько тысяч деревьев решений, которые необходимо строить последовательно. Построение решающего дерева на выборках типичного размера и современном железе, даже с учетом всех оптимизаций, требует небольшого, но всё-таки заметного времени (0.1-1с), которое для всего ансамбля превратится в десятки минут. Это не так быстро, как обучение линейных моделей, но всё-таки значительно быстрее, чем обучение типичных нейросетей.

Темп обучения

Обучение композиции с помощью градиентного бустинга может привести к переобучению, если базовые алгоритмы слишком сложные. Например, если сделать решающие деревья слишком глубокими (более 10 уровней), то при обучении бустинга ошибка на обучающей выборке даже при довольно скромном K может приблизиться к нулю, то есть предсказание будет почти идеальным, но на тестовой выборке всё будет плохо.

Существует два решения этой проблемы. Во-первых, необходимо упростить базовую модель, уменьшив глубину дерева (либо примерив какие-либо ещё техники регуляризации). Во-вторых, мы можем ввести параметр, называемый темпом обучения (learning rate) $\eta \in (0,1]$:

$$a_{k+1}(x) = a_k(x) + \eta b_{k+1}(x)$$

Присутствие этого параметра означает, что каждый базовый алгоритм вносит относительно небольшой вклад во всю композицию: если расписать сумму целиком, она будет иметь вид

$$a_{k+1}(x) = b_1(x) + \eta b_2(x) + \eta b_3(x) + \cdots + \eta b_{k+1}(x)$$

Значение параметра обычно определяется эмпирически по входным данным. В библиотеке CatBoost темп обучения может быть выбран автоматически по набору данных. Для этого используется заранее обученная линейная модель, предсказывающая темп обучения по мета-параметрам выборки данных: числу объектов, числу признаков и другим.

Темп обучения связан с количеством итераций градиентного бустинга. Чем меньше темп обучения, тем больше итераций потребуется сделать для достижения того же качества на обучающей выборке.

Важность функции

Отдельные деревья решений можно легко интерпретировать, просто визуализируя их структуру. Но в модели градиентного бустинга содержатся сотни деревьев, и поэтому её нелегко интерпретировать путем визуализации входящих в неё деревьев. При этом хотелось бы, как минимум, понимать, какие именно признаки в данных оказывают наибольшее влияние на предсказание композиции.

Можно сделать следующее наблюдение: признаки, используемые в верхней части дерева, влияют на окончательное предсказание для большей доли обучающих объектов, чем признаки, попавшие на более глубокие уровни. Таким образом, ожидаемая доля обучающих объектов, для которых происходило ветвление по данному признаку, может быть использована в качестве оценки его относительной важности для итогового предсказания. Усредняя полученные оценки важности признаков по всем решающим деревьям из ансамбля, можно

уменьшить дисперсию такой оценки и использовать ее для отбора признаков. Этот метод известен как MDI (mean decrease in impurity - среднее уменьшение примеси). Существуют и другие методы оценки важности признаков для ансамблей: например, permutation feature importance (важность функции перестановки) и множество разных подходов, предлагаемых в библиотеке CatBoost. Все эти техники отбора признаков применимы также и для случайных лесов.

Пример градиентного бустинга

Есть набор данных продажи дома. Здесь возраст, площадь, расположение – независимые переменные, а цена - зависимая переменная или целевая переменная.

Таблица 1.

возраст	площадь	расположение	цена
5	1500	5	480
11	2030	12	1090
14	1442	6	350
8	2501	4	1310
12	1300	9	400
10	1789	11	500

Шаг 1: Рассчитать среднее/среднее значение целевой переменной.

$$\text{Средняя цена} = \frac{480 + 1090 + 350 + 1310 + 400 + 500}{6} = 688$$

Таблица 2.

возраст	площадь	расположение	цена	средняя цена
5	1500	5	480	688
11	2030	12	1090	688
14	1442	6	350	688
8	2501	4	1310	688
12	1300	9	400	688
10	1789	11	500	688

Шаг 2: Рассчитайте остатки для каждого образца.

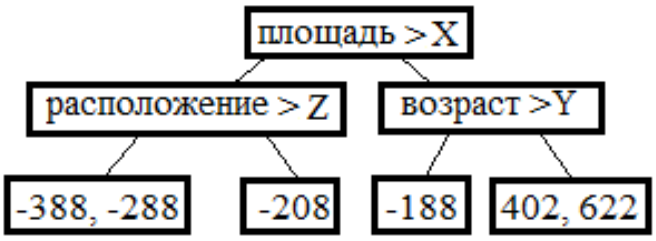
$$\text{Остаток} = \text{Реальная стоимость} - \text{Прогнозируемое значение}$$

Таблица 3.

возраст	площадь	расположение	цена	средняя цена	остаток
5	1500	5	480	688	-208

11	2030	12	1090	688	402
14	1442	6	350	688	-338
8	2501	4	1310	688	622
12	1300	9	400	688	-288
10	1789	11	500	688	-188

Шаг 3: Построить дерево решений. Строим дерево с целью предсказания остатков.

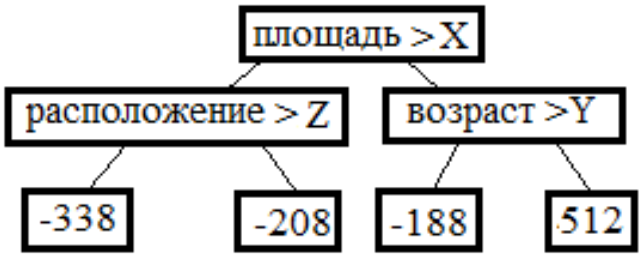


В случае, если остатков больше, чем листовых узлов (здесь их 6 остатков), некоторые остатки окажутся внутри одного и того же листа. Когда это происходит, вычисляется их среднее значение и помещаем его внутрь листа.

$$\frac{(-388) + (-288)}{2} = -338$$

$$\frac{402 + 622}{2} = 512$$

После этого дерево станет таким.



Шаг 4: Предсказать целевую метку, используя все деревья в ансамбле.

Каждая выборка проходит через решающие узлы вновь сформированного дерева, пока не достигнет заданного опережения. Остаток в указанном листе используется для прогнозирования цены дома.

Прогнозируемая цена = Средняя цена + Скорость обучения * Остаток предсказанный деревом решений

Тогда расчет для остаточного значения (-338) и (-208) на шаге 2

$$\text{Прогнозируемая цена}_{-338} = 688 + 0{,}1 * (-338) = 654{,}2$$

$$\text{Прогнозируемая цена}_{-208} = 688 + 0{,}1 * (-208) = 667{,}2$$

Изначально была взята 0,1 в качестве скорости обучения.

Точно так же рассчитается прогнозируемую цена для других значений.

Шаг 5: Вычислите новые остатки

$$\text{Остаток} = \text{Реальная стоимость} - \text{Прогнозируемое значение}$$

При цене 350 и 480 соответственно.

$$\text{Остаток}_{350} = 350 - 654{,}2 = -304{,}2$$

$$\text{Остаток}_{480} = 480 - 667{,}2 = -187{,}2$$

С нашим деревом решений получили следующие новые остатки.

Таблица 4.

возраст	площадь	расположение	цена	средняя цена	остаток	Новый остаток
5	1500	5	480	688	-208	-187,2
11	2030	12	1090	688	402	350,8
14	1442	6	350	688	-338	-304,2
8	2501	4	1310	688	622	570,8
12	1300	9	400	688	-288	-254,1
10	1789	11	500	688	-188	-169,2

Шаг 6: Повторяем шаги с 3 по 5, пока количество итераций не совпадет с числом, указанным в гиперпараметре (количество оценок).

Шаг 7: После обучения используйте все деревья в ансамбле, чтобы сделать окончательный прогноз относительно значения целевой переменной. Окончательный прогноз будет равен среднему значению, которое вычисляется на шаге 1, плюс все остатки, предсказанные деревьями, составляющими лес, умноженные на скорость обучения.

Тогда ДР – дерево решений

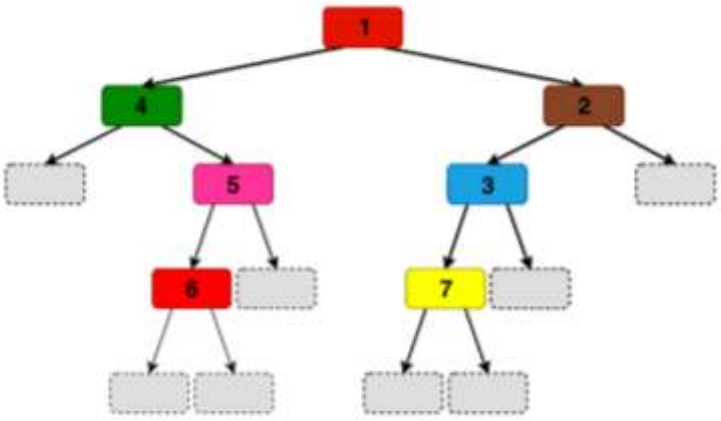
Прогнозируемая цена = Средняя цена + Скорость обучения * Остаток предсказанный ДР_1 + Скорость обучения * Остаток предсказанный ДР_2 + … + Скорость обучения * Остаток предсказанный ДР_N

Реализации

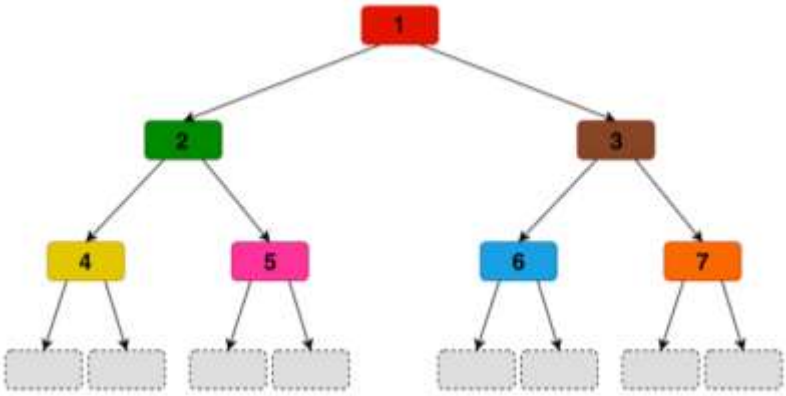
Хороших реализаций градиентного бустинга есть, как минимум, три: LightGBM, XGBoost и CatBoost. Исторически они отличались довольно сильно, но за последние годы успели скопировать друг у друга все хорошие идеи.

Одним из основных отличий LightGBM, XGBoost и CatBoost является форма решающих деревьев. LightGBM строит деревья по принципу: «На каждом шаге делим вершину с наилучшим градиентом», а основным критерием остановки выступает максимально допустимое количество вершин в дереве. Это приводит к тому, что деревья получаются несимметричными, то есть поддеревья могут иметь разную глубину – например, левое поддерево может иметь глубину 2, а правое может разрастись до глубины 15. С одной стороны, это позволяет быстро

подогнаться под обучающие данные. С другой, бесконтрольный рост дерева в глубину неизбежно ведет к переобучению, поэтому LightGBM позволяет помимо количества вершин ограничивать и максимальную глубину. Впрочем, это ограничение обычно все равно выше, чем для XGBoost и CatBoost.

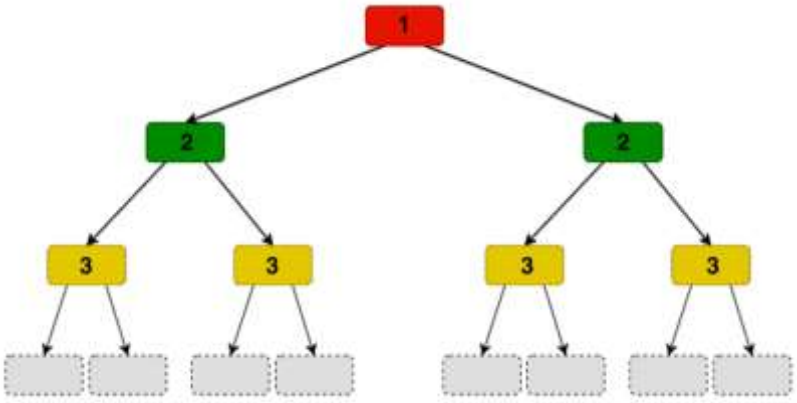


XGBoost строит деревья по принципу: «Строим дерево последовательно по уровням до достижения максимальной глубины». Отдельного ограничения на количество вершин нет, так как оно естественным образом получается из ограничения на глубину дерева. В XGBoost деревья «стремятся» быть симметричными по глубине, и в идеале получается полное бинарное дерево, если это не противоречит другим ограничениям (например, ограничению на минимальное количество объектов в листе). Такие деревья обычно являются более устойчивыми к переобучению.



CatBoost строит деревья по принципу: «Все вершины одного уровня имеют одинаковый предикат». Одинаковые сплиты (разделение) во всех вершинах одного уровня позволяют избавиться от ветвлений (конструкций if-else) в коде инференса модели с помощью битовых операций и получить более эффективный код, который в разы ускоряет применение модели, в особенности в случае применения на батчах (пакетный файл). Кроме этого, такое ограничение на форму дерева выступает в качестве сильной регуляризации, что делает модель более устойчивой к переобучению. Основным критерием остановки, как и в случае

XGBoost, является ограничение на глубину дерева. Однако, в отличие от XGBoost, в CatBoost всегда создаются полные бинарные деревья, несмотря на то, что в некоторые поддеревья может не попасть ни одного объекта из обучающей выборки.



Преимущества градиентного бустинга

1. В большинстве случаев предсказуемая точность алгоритма повышения градиента выше.
2. Он обеспечивает большую гибкость и может оптимизировать различные функции потерь, а также предоставляет несколько вариантов настройки гиперпараметров, которые делают функцию очень гибкой.
3. В большинстве случаев предварительная обработка данных не требуется.
4. Алгоритм Gradient Boosting отлично работает с категориальными и числовыми данными.
5. Обрабатывает отсутствующие данные — вменение отсутствующих значений не требуется.

Недостатки градиентного бустинга

1. Модели повышения градиента будут продолжать совершенствоваться, чтобы свести к минимуму все ошибки. Это может чрезмерно подчеркнуть выбросы и привести к переподгонке. Необходимо использовать перекрестную проверку для нейтрализации.
2. Это очень дорого в вычислительном отношении — для GBM часто требуется много деревьев (> 1000), которые могут быть исчерпывающими по времени и памяти.
3. Высокая гибкость приводит к тому, что многие параметры взаимодействуют и сильно влияют на поведение подхода (количество итераций, глубина дерева, параметры регуляризации и т. д.). Это требует поиска по большой сетке во время настройки.