Семинары по композиционным методам

Евгений Соколов
sokolov.evg@gmail.com

17 марта 2016 г.

2 Композиционные методы машинного обучения

§2.1 Градиентный бустинг

Ранее мы изучили алгоритм AdaBoost, строящий композицию алгоритмов путем последовательной минимизации экспоненциальной функции потерь. Из плюсов этого подхода можно отметить аналитическую запись формул для поиска базовых алгоритмов, гарантии сходимости, направленность на максимизацию отступов. Минусы вытекают из вида экспоненциальной функции потерь — алгоритм неустойчив к выбросам, а также крайне тяжело адаптируем для других задач (регрессии, многоклассовой классификации). Ниже мы рассмотрим другой подход к построению композиций — градиентный бустинг, предложенный Фридманом [1]. Он работает для любых дифференцируемых функций потерь и является одним из наиболее мощных и универсальных на сегодняшний день.

2.1.1 Бустинг в задаче регрессии

Рассмотрим задачу минимизации квадратичной функции потерь:

$$\frac{1}{2}\sum_{i=1}^{\ell}(a(x_i) - y_i)^2 \to \min_a$$

Будем искать алгоритм в виде суммы базовых:

$$a_N(x) = \sum_{n=1}^{N} b_n(x),$$

где базовые алгоритмы b_n принадлежат некоторому семейству $\mathcal{A}$.

Построим первый базовый алгоритм:

$$b_1(x) := \arg\min_{b \in \mathcal{A}} \frac{1}{2}\sum_{i=1}^{\ell}(b(x_i) - y_i)^2$$

Решение такой задачи не представляет трудностей для многих семейств алгоритмов. Теперь мы можем посчитать остатки на каждом объекте — расстояния от ответа нашего алгоритма до истинного ответа:

$$s_i^{(1)} = y_i - b_1(x_i)$$

Если прибавить эти остатки к ответам построенного алгоритма, то он не будет допускать ошибок на обучающей выборке. Значит, будет разумным построить второй алгоритм так, чтобы его ответы были как можно ближе к остаткам:

$$b_2(x) := \arg\min_{b \in \mathcal{A}} \frac{1}{2} \sum_{i=1}^{\ell} (b(x_i) - s_i^{(1)})^2$$

Каждый следующий алгоритм тоже будем настраивать на остатки предыдущих:

$$s_i^{(N)} = y_i - \sum_{n=1}^{N-1} b_n(x_i) = y_i - a_{N-1}(x_i), \qquad i = 1, \ldots, \ell;$$

$$b_N(x) := \arg\min_{b \in \mathcal{A}} \frac{1}{2} \sum_{i=1}^{\ell} (b(x_i) - s_i^{(N)})^2$$

Описанный метод прост в реализации, хорошо работает и может быть найден во многих библиотеках — например, в `scikit-learn`.

Заметим, что остатки могут быть найдены как антиградиент функции потерь по ответу модели, посчитанный в точке ответа уже построенной композиции:

$$s_i^{(N)} = y_i - a_{N-1}(x_i) = -\frac{\partial}{\partial z} \frac{1}{2} \sum_{i=1}^{\ell} (z - y_i)^2 \Bigg|_{z = a_{N-1}(x_i)}$$

Получается, что выбирается такой базовый алгоритм, который как можно сильнее уменьшит ошибку композиции — это свойство вытекает из его близости к антиградиенту функционала на обучающей выборке. Попробуем разобраться с этим свойством подробнее, а также попробовать обобщить его на другие функции потерь.

2.1.2 Градиентный бустинг

Пусть дана некоторая дифференцируемая функция потерь $L(y, z)$. Будем строить взвешенную сумму базовых алгоритмов:

$$a_N(x) = \sum_{n=0}^{N} \gamma_n b_n(x)$$

Заметим, что в композиции имеется начальный алгоритм $b_0(x)$. Как правило, коэффициент γ_0 при нем берут равным единице, а сам алгоритм выбирают очень простым, например:

- нулевым $b_0(x) = 0$;
- возвращающим самый популярный класс (в задачах классификации):

$$b_0(x) = \arg\max_{y \in \mathbb{Y}} \sum_{i=1}^{\ell} [y_i = y]$$

- возвращающим средний ответ (в задачах регрессии):

$$b_0(x) = \frac{1}{\ell}\sum_{i=1}^{\ell} y_i$$

Допустим, мы построили композицию $a_{N-1}(x)$ из $N-1$ алгоритма, и хотим выбрать следующий базовый алгоритм $b_N(x)$ так, чтобы как можно сильнее уменьшить ошибку:

$$\sum_{i=1}^{\ell} L(y_i, a_{N-1}(x_i) + \gamma_N b_N(x_i)) \to \min_{b_N, \gamma_N}$$

Ответим в первую очередь на следующий вопрос: если бы в качестве алгоритма $b_N(x)$ мы могли выбрать совершенно любую функцию, то какие значения ей следовало бы принимать на объектах обучающей выборки? Иными словами, нам нужно понять, какие числа $s_1, \ldots, s_\ell$ надо выбрать для решения следующей задачи:

$$\sum_{i=1}^{\ell} L(y_i, a_{N-1}(x_i) + s_i) \to \min_{s_1,\ldots,s_\ell}$$

Понятно, что можно требовать $s_i = y_i - a_{N-1}(x_i)$, но такой подход никак не учитывает особенностей функции потерь $L(y, z)$ и требует лишь точного совпадения предсказаний и истинных ответов. Более разумно потребовать, чтобы сдвиг s_i был противоположен производной функции потерь в точке $z = a_{N-1}(x_i)$:

$$s_i = -\left.\frac{\partial L}{\partial z}\right|_{z=a_{N-1}(x_i)}$$

В этом случае мы сдвинемся в сторону убывания функции потерь. Заметим, что вектор сдвигов $s = (s_1, \ldots, s_\ell)$ совпадает с антиградиентом:

$$\left(-\left.\frac{\partial L}{\partial z}\right|_{z=a_{N-1}(x_i)}\right)_{i=1}^{\ell} = -\nabla_s \sum_{i=1}^{\ell} L(y_i, a_{N-1}(x_i) + s_i)$$

При таком выборе сдвигов s_i мы, по сути, сделаем один шаг градиентного спуска, двигаясь в сторону наискорейшего убывания ошибки на обучающей выборке. Отметим, что речь идет о градиентном спуске в ℓ-мерном пространстве предсказаний алгоритма на объектах обучающей выборки. Поскольку вектор сдвига будет свой на каждой итерации, правильнее обозначать его как $s_i^{(N)}$, но для простоты будем иногда опускать верхний индекс.

Итак, мы поняли, какие значения новый алгоритм должен принимать на объектах обучающей выборки. По данным значениям в конечном числе точек необходимо построить функцию, заданную на всем пространстве объектов. Это классическая задача обучения с учителем, которую мы уже хорошо умеем решать. Один из самых простых функционалов — среднеквадратичная ошибка. Воспользуемся им для поиска базового алгоритма, приближающего градиент функции потерь на обучающей выборке:

$$b_N(x) = \arg\min_{b \in \mathcal{A}} \sum_{i=1}^{\ell} (b(x_i) - s_i)^2$$

Отметим, что здесь мы оптимизируем квадратичную функцию потерь независимо от функционала исходной задачи — вся информация о функции потерь L находится в антиградиенте s_i, а на данном шаге лишь решается задача аппроксимации функции по ℓ точкам. Разумеется, можно использовать и другие функционалы, но среднеквадратичной ошибки, как правило, оказывается достаточно.

После того, как новый базовый алгоритм найден, можно подобрать коэффициент при нем по аналогии с наискорейшим градиентным спуском:

$$\gamma_N = \arg\min_{\gamma \in \mathbb{R}} \sum_{i=1}^{\ell} L(y_i, a_{N-1}(x_i) + \gamma b_N(x_i))$$

Описанный подход с аппроксимацией антиградиента базовыми алгоритмами и называется градиентным бустингом. Данный метод представляет собой поиск лучшей функции, восстанавливающей истинную зависимость ответов от объектов, в пространстве всех возможных функций. Ищем мы данную функцию с помощью «псевдоградиентного» спуска — каждый шаг делается вдоль направления, задаваемого некоторым базовым алгоритмом. При этом сам базовый алгоритм выбирается так, чтобы как можно лучше приближать антиградиент ошибки на обучающей выборке.

2.1.3 Регуляризация

Сокращение шага. На практике оказывается, что градиентный бустинг очень быстро строит композицию, ошибка которой на обучении выходит на асимптоту, после чего начинает настраиваться на шум и переобучаться. Это явление можно объяснить одной из двух причин:

- Если базовые алгоритмы очень простые (например, решающие деревья небольшой глубины), то они плохо приближают вектор антиградиента. По сути, добавление такого базового алгоритма будет соответствовать шагу вдоль направления, сильно отличающегося от направления наискорейшего убывания. Соответственно, градиентный бустинг может свестись к случайному блужданию в пространстве.

- Если базовые алгоритмы сложные (например, глубокие решающие деревья), то они способны за несколько шагов бустинга идеально подогнаться под обучающую выборку — что, очевидно, будет являться переобучением, связанным с излишней сложностью семейства алгоритмов.

Хорошо зарекомендовавшим себя способом решения данной проблемы является *сокращение шага*: вместо перехода в оптимальную точку в направлении антиградиента делается укороченный шаг

$$a_N(x) = a_{N-1}(x) + \eta \gamma_N b_N(x),$$

где $\eta \in (0, 1]$ — темп обучения [1]. Как правило, чем меньше темп обучения, тем лучше качество итоговой композиции. Сокращение шага, по сути, позволяет понизить доверие к направлению, восстановленному базовым алгоритмом.

Также следует обратить внимание на число итераций градиентного бустинга. Хотя ошибка на обучении монотонно стремится к нулю, ошибка на контроле, как

правило, начинает увеличиваться после определенной итерации. Оптимальное число итераций можно выбирать, например, по отложенной выборке или с помощью кросс-валидации.

Стохастический градиентный бустинг. Еще одним способом улучшения качества градиентного бустинга является внесение рандомизации в процесс обучения базовых алгоритмов [2]. А именно, алгоритм b_N обучается не по всей выборке X^ℓ, а лишь по ее случайному подмножеству $X^k \subset X^\ell$. В этом случае понижается уровень шума в обучении, а также повышается эффективность вычислений. Существует рекомендация брать подвыборки, размер которых вдвое меньше исходной выборки.

2.1.4 Функции потерь

Регрессия. При вещественном целевом векторе, как правило, используют квадратичную функцию потерь, формулы для которой уже были приведены в разделе 2.1.1. Другой вариант — модуль отклонения $L(y, z) = |y - z|$, для которого антиградиент вычисляется по формуле

$$s_i^{(N)} = -\operatorname{sign}(a_{N-1}(x_i) - y_i).$$

Классификация. В задаче классификации с двумя классами $\mathbb{Y} = \{+1, -1\}$ разумным выбором является настройка функции $p_+(x) \in [0, 1]$, возвращающей вероятность класса $+1$. В этом случае мы можем измерить правдоподобие обучающей выборки при условии модели $p_+(x)$:

$$P(p_+) = \prod_{i=1}^{\ell} p_+(x_i)^{[y_i=1]}(1 - p_+(x_i))^{[y_i=-1]}.$$

Данное правдоподобие следует максимизировать. Гораздо удобнее минимизировать отрицательный логарифм правдоподобия:

$$-\sum_{i=1}^{\ell} ([y_i = 1] \log p_+(x_i) + [y_i = -1] \log(1 - p_+(x_i))) \to \min_{p_+(x)}. \tag{2.1}$$

Такая задача крайне неудобна — нам нужно искать алгоритм $p_+(x)$ с ограничением, что его ответ лежит на отрезке $[0, 1]$. Будем вместо этого искать алгоритм $a(x) \in \mathbb{R}$, возвращающий любые вещественные числа, который связан с вероятностью $p_+(x)$ через сигмоидную функцию:

$$p_+(x) = \frac{1}{1 + \exp(-a(x))}$$

Соответственно, вероятность отрицательного класса задается формулой

$$p_-(x) = 1 - p_+(x) = \frac{1}{1 + \exp(a(x))}$$

Подставляя эти выражения в логарифм правдоподобия (2.1), получаем:

$$-\sum_{i=1}^{\ell}\left(-[y_i=1]\log(1+\exp(-a(x_i)))-[y_i=-1]\log(1+\exp(a(x_i)))\right)=$$
$$=\sum_{i=1}^{\ell}\log(1+\exp(-y_i a(x_i))).$$

Мы получили логистическую функцию потерь, с которой уже сталкивались при изучении линейных методов:

$$L(y,z)=\log(1+\exp(-yz)).$$

Легко показать, что алгоритм возвращает логарифм отношения оценок вероятностей классов:

$$a(x)=\frac{1}{2}\log\frac{p_+(x)}{1-p_+(x)}.$$

Задача 2.1. *Как будет выглядеть задача поиска базового алгоритма $b_N(x)$ в случае с логистической функцией потерь?*

Решение. Найдем компоненты антиградиента s_i:

$$s_i^{(N)}=-\left.\frac{\partial L(y_i,z)}{\partial z}\right|_{z=a_{N-1}(x_i)}=\frac{y_i}{1+\exp(y_i a_{N-1}(x_i))}. \tag{2.2}$$

Значит, задача поиска базового алгоритма примет вид

$$b_N=\arg\min_{b\in\mathcal{A}}\sum_{i=1}^{\ell}\left(b(x_i)-\frac{y_i}{1+\exp(y_i a_{N-1}(x_i))}\right)^2.$$

■

Логистическая функция потерь имеет интересную особенность, связанную со взвешиванием объектов. Заметим, что ошибка на N-й итерации может быть записана как

$$Q(a_N)=\sum_{i=1}^{\ell}\log\left(1+\exp(-y_i a_N(x_i))\right)=$$
$$=\sum_{i=1}^{\ell}\log\left(1+\exp(-y_i a_{N-1}(x_i))\exp(-y_i\gamma_N b_N(x_i))\right).$$

Если отступ $y_i a_{N-1}(x_i)$ на i-м объекте большой положительный, то данный объект не будет вносить практически никакого вклада в ошибку, и может быть исключен из всех вычислений на текущей итерации без потерь. Таким образом, величина

$$w_i^{(N)}=\exp(-y_i a_{N-1}(x_i))$$

может служить мерой важности объекта x_i на N-й итерации градиентного бустинга.

2.1.5 Градиентный бустинг над деревьями

Считается, что градиентный бустинг над решающими деревьями — один из самых универсальных и сильных методов машинного обучения, известных на сегодняшний день. В частности, на градиентном бустинге над деревьями основан MatrixNet — алгоритм ранжирования компании Яндекс [3].

Вспомним, что решающее дерево разбивает все пространство на непересекающиеся области, в каждой из которых его ответ равен константе:

$$b_n(x) = \sum_{j=1}^{J} b_{nj}[x \in R_j],$$

где $j = 1, \ldots, J$ — индексы листьев, R_j — соответствующие области разбиения, b_{nj} — значения в листьях. Значит, на N-й итерации бустинга композиция обновляется как

$$a_N(x) = a_{N-1}(x) + \gamma_N \sum_{j=1}^{J} b_{Nj}[x \in R_j].$$

Видно, что добавление в композицию одного дерева с J листьями равносильно добавлению J базовых алгоритмов, представляющих собой предикаты. Мы можем улучшить качество композиции, подобрав свой коэффициент при каждом из предикатов:

$$\sum_{i=1}^{\ell} L\left(y_i, a_{N-1}(x_i) + \sum_{j=1}^{J} \gamma_{Nj}[x \in R_j]\right) \to \min_{\{\gamma_{Nj}\}_{j=1}^{J}}.$$

Поскольку области разбиения R_j не пересекаются, данная задача распадается на J независимых подзадач:

$$\gamma_{Nj} = \arg\min_{\gamma} \sum_{x_i \in R_j} L(y_i, a_{N-1}(x_i) + \gamma), \qquad j = 1, \ldots, J.$$

В некоторых случаях оптимальные коэффициенты могут быть найдены аналитически.

Задача 2.2. *Найдите оптимальные коэффициенты $\{\gamma_{Nj}\}_{j=1}^{J}$ для функционалов квадратичной и абсолютной ошибки.*

Решение. Требуется решить задачи

$$\sum_{x_i \in R_j} (a_{N-1}(x_i) + \gamma - y_i)^2 \to \min_{\gamma}$$

и

$$\sum_{x_i \in R_j} |a_{N-1}(x_i) + \gamma - y_i| \to \min_{\gamma}.$$

Известно, что решениями данных задач являются среднее и медиана остатков:

$$\gamma_1 = \frac{1}{|R_j|} \sum_{x_i \in R_j} (y_i - a_{N-1}(x_i));$$

$$\gamma_2 = \operatorname*{median}_{x_i \in R_j} \{y_i - a_{N-1}(x_i)\}.$$

Видно, что в случае с квадратичным функционалом дополнительную настройку коэффициентов можно не делать — деревья и так настраиваются на квадратичный функционал, и в листьях будет записано среднее значение ответа по попавшим в них объектам.

В случае с абсолютной функцией потерь настройка коэффициентов уже несет в себе пользу. Сами деревья настраиваются так, чтобы квадратичное отклонение от знака ошибки было минимально (т.е. минимизируется $(b(x) - \text{sign}(y_i - a_{N-1}(x_i)))^2$), но затем мы изменяем значения в листьях так, чтобы они были оптимальны с точки зрения модуля отклонения. Безусловно, дерево можно сразу настраивать на модули отклонений, но такая процедура работает гораздо медленнее, чем настройка на квадратичные потери.

■

Рассмотрим теперь логистическую функцию потерь. В этом случае нужно решить задачу

$$F_j^{(N)}(\gamma) = \sum_{x_i \in R_j} \log\left(1 + \exp\left(-y_i(a_{N-1}(x_i) + \gamma)\right)\right) \to \min_{\gamma}.$$

Данная задача может быть решена лишь с помощью итерационных методов, аналитической записи для оптимального γ не существует. Однако на практике обычно нет необходимости искать точное решение — оказывается достаточным сделать лишь один шаг метода Ньютона-Рафсона из начального приближения $\gamma_{Nj} = 0$. Можно показать, что в этом случае

$$\gamma_{Nj} = \frac{\partial F_j^{(N)}(0)}{\partial \gamma} \Bigg/ \frac{\partial^2 F_j^{(N)}(0)}{\partial \gamma^2} = -\sum_{x_i \in R_j} s_i^{(N)} \Bigg/ \sum_{x_i \in R_j} |s_i^{(N)}|(1 - |s_i^{(N)}|).$$

2.1.6 Связь с AdaBoost

Попробуем применить градиентный бустинг к экспоненциальной функции потерь, на оптимизации которой основан AdaBoost:

$$L(a, X^\ell) = \sum_{i=1}^{\ell} \exp\left(-y_i \sum_{n=1}^{N} \gamma_n b_n(x_i)\right).$$

Найдем компоненты ее антиградиента после $(N-1)$-й итерации:

$$s_i = -\left.\frac{\partial L(y_i, z)}{\partial z}\right|_{z = a_{N-1}(x_i)} = y_i \underbrace{\exp\left(-y_i \sum_{n=1}^{N-1} \gamma_n b_n(x_i)\right)}_{w_i}.$$

Заметим, что антиградиент представляет собой ответ на объекте, умноженный на его вес (такой же, как в AdaBoost). Если все веса будут равны единице, то следующий базовый классификатор будет просто настраиваться на исходный целевой вектор $(y_i)_{i=1}^{\ell}$; штраф за выдачу ответа, противоположного правильному, будет равен 4 (поскольку при настройке базового алгоритма используется квадратичная

функция потерь). Если же какой-либо объект будет иметь большой отступ, то его вес окажется близким к нулю, и штраф за выдачу любого ответа будет равен 1.

Отметим, что многие функционалы ошибки классификации выражаются через отступы объектов:

$$L(a_{N-1}, X^\ell) = \sum_{i=1}^{\ell} L(a_{N-1}(x_i), y_i) = \sum_{i=1}^{\ell} \tilde{L}(y_i a_{N-1}(x_i)).$$

В этом случае антиградиент принимает вид

$$s_i = y_i \underbrace{\left(-\frac{\partial \tilde{L}(y_i a_{N-1}(x_i))}{\partial a_{N-1}(x_i)} \right)}_{w_i},$$

то есть тоже взвешивает ответы с помощью ошибки на них.

2.1.7 Влияние шума на обучение.

Выше мы находили формулу для антиградиента при использовании экспоненциальной функции потерь:

$$s_i = y_i \underbrace{\exp\left(-y_i \sum_{n=1}^{N-1} \gamma_n b_n(x_i) \right)}_{w_i}.$$

Заметим, что если отступ на объекте большой и отрицательный (что обычно наблюдается на шумовых объектах), то вес становится очень большим, причем он никак не ограничен сверху. В результате базовый классификатор будет настраиваться исключительно на шумовые объекты, что может привести к неустойчивости его ответов и переобучению.

Рассмотрим теперь логистическую функцию потерь, которая также может использоваться в задачах классификации:

$$L(a, X^\ell) = \sum_{i=1}^{\ell} \log\left(1 + \exp\left(-y_i a(x_i)\right)\right).$$

Найдем ее антиградиент после $(N-1)$-го шага:

$$s_i = y_i \underbrace{\frac{1}{1 + \exp(y_i a_{N-1}(x_i))}}_{= w_i^{(N)}}.$$

Теперь веса ограничены сверху единицей. Если отступ на объекте большой отрицательный (то есть это выброс), то вес при нем будет близок к единице; если же отступ на объекте близок к нулю (то есть это объект, на котором классификация неуверенная, и нужно ее усиливать), то вес при нем будет примерно равен $1/2$. Таким образом, вес при шумовом объекте будет всего в два раза больше, чем вес при нормальных объектах, что не должно сильно повлиять на процесс обучения.

§2.2 Методы оптимизации второго порядка

Как мы выяснили выше, градиентный бустинг осуществляет градиентный спуск в пространстве прогнозов алгоритма на обучающей выборке. Здесь может возникнуть вполне логичный вопрос: а почему бы не воспользоваться другим, более эффективным методом оптимизации? Наиболее явными кандидатами являются методы оптимизации второго порядка — например, метод Ньютона. При оптимизации числовой функции $Q(w)$ шаг в методе Ньютона осуществляется по формуле

$$w^{(n)} = w^{(n-1)} - H^{-1}(w^{(n-1)})\nabla_w Q(w^{(n-1)}),$$

где $H(w)$ — матрица вторых производных, которая также называется матрицей Гессе.

Этим же подходом можно воспользоваться и в градиентном бустинге. Нам нужно как можно сильнее уменьшить значение следующей функции путем подбора сдвигов s_i:

$$Q(s) = \sum_{i=1}^{\ell} L(y_i, a_{N-1}(x_i) + s_i)$$

Мы уже находили вектор градиента:

$$\nabla_s Q(s) = g = \left(\frac{\partial L}{\partial z}\bigg|_{z=a_{N-1}(x_i)} \right)_{i=1}^{\ell}$$

Заметим, что матрица вторых производных тут будет диагональной, поскольку каждая переменная s_i входит лишь в одно отдельное слагаемое:

$$H = \operatorname{diag}\left(\frac{\partial^2 L}{\partial z^2}\bigg|_{z=a_{N-1}(x_1)}, \ldots, \frac{\partial^2 L}{\partial z^2}\bigg|_{z=a_{N-1}(x_\ell)} \right)$$

Мы здесь пользуемся функционалом, который представляет собой сумму ошибок на всех объектах. Такое представление подходит во многих задачах, но не является максимально общим. Дело в том, что в некоторых ситуациях необходимо использовать функционалы, которые измеряют качество сортировки объектов алгоритмом — это, иными словами, функционалы качества ранжирования. Их особенность заключается как раз в том, что матрица вторых производной уже не будет диагональной.

Зная градиент и матрицу Гессе, мы можем выписать формулу для сдвигов s:

$$s = -H^{-1}g$$

Поскольку обращение матрицы является неустойчивой операцией, правильнее будет находить вектор сдвигов через систему линейных уравнений

$$Hs = -g$$

В случае с диагональной матрицей Гессе данный подход сводится к домножению каждой компоненты вектора антиградиента на некоторые коэффициенты. В общем же случае такой подход может оказаться слишком сложным, поскольку при больших

размерах выборки матрица Гессе будет получаться слишком большой для эффективной работы. После того, как рассчитан вектор сдвигов, процедура будет такой же, как и раньше — обучаем алгоритм на данные сдвиги, находим коэффициент при нем, добавляем в композицию.

На похожей идее основан метод LogitBoost, который настраивает композицию с использованием логистической функции потерь, исходя из несколько иных предположений. Использование метода Ньютона приводит к тому, что базовый алгоритм настраивается на взвешенный функционал, что может затруднять обучение. Более того, формулы для весов получаются не вполне устойчивыми, и нередко в них происходит деление на очень маленькое число. Чтобы избежать этого, вводится ряд достаточно грубых эвристик.

Обратим внимание, что трюк с переподбором прогнозов в листьях базовых решающих деревьев похож на применение метода Ньютона. Допустим, мы как-то выбрали сдвиги s_i и обучили на них решающее дерево $b_n(x)$. После этого на объекте x_i обучающей выборки будет сделан сдвиг $b_n(x_i)$. Сдвиги будут одинаковыми на тех объектах, которые попали в один и тот же лист дерева. Если сделать переподбор, то сдвиги будут изменены так, чтобы как можно сильнее уменьшать исходный функционал ошибки. По сути, благодаря этому сдвиги подбираются индивидуально под группы объектов, что близко к использованию методов второго порядка с диагональной матрицей Гессе — там тоже выставляются индивидуальные коэффициенты при компонентах сдвига.

Список литературы

[1] *Friedman, Jerome H.* (2001). Greedy Function Approximation: A Gradient Boosting Machine. // Annals of Statistics, 29(5), p. 1189–1232.

[2] *Friedman, Jerome H.* (1999). Stochastic Gradient Boosting. // Computational Statistics and Data Analysis, 38, p. 367–378.

[3] *Gulin, A., Karpovich, P.* (2009). Greedy function optimization in learning to rank. http://romip.ru/russir2009/slides/yandex/lecture.pdf