

6. Градиентный бустинг

Цель занятия

В результате обучения на этой неделе вы:

- погрузитесь в изучение градиентного бустинга
- познакомитесь с алгоритмом AdaBoost
- рассмотрите визуализацию градиентного бустинга, посмотрите, как строится алгоритм градиентного бустинга над решающими деревьями
- научитесь решать задачи классификации и регрессии с помощью градиентного бустинга
- познакомитесь с библиотекой CatBoost

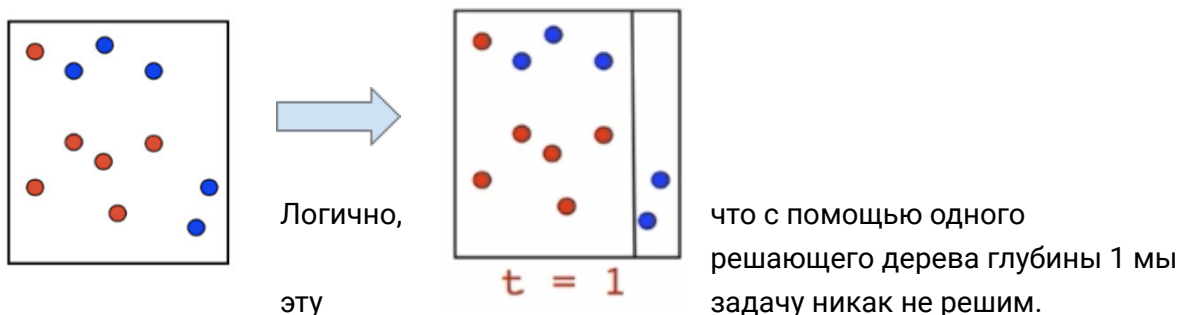
План занятия

1. [Бустинг](#)
2. [Градиентный бустинг](#)
3. [Визуализация градиентного бустинга](#)
4. [CatBoost](#)

Конспект занятия

1. Бустинг

Построим ансамбль из нескольких простых моделей. Рассмотрим для примера “решающий пен” – решающее дерево глубины 1. По сути это условие “if” – налево или направо. Мы имеем право только один раз разделить выборку на две половинки.



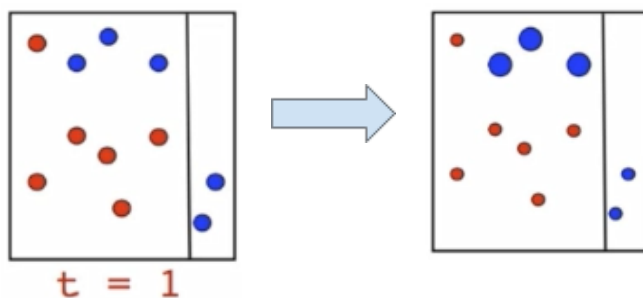
Но, используя бустинг, мы решим эту задачу с помощью решающих пеней в качестве базовых алгоритмов.

Простое усреднение в данном методе использовать нельзя. Усреднение – линейная комбинация, решающий пень – линейная модель. Линейная комбинация линейных моделей ни к чему не приведет.

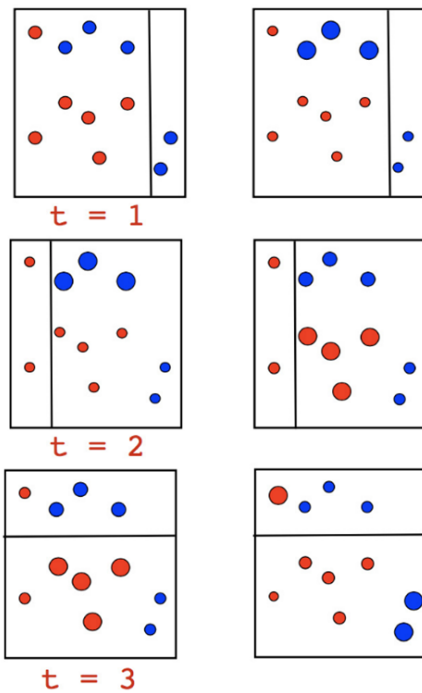
Если у нас ансамбль из решающих пней, можно попросить каждый следующий элемент ансамбля не усредняться со всеми предыдущими, а пытаться исправить ошибки, которые допустили предыдущие элементы. То есть будем строить ансамбль не параллельно, а последовательно. И каждый следующий элемент ансамбля будет уменьшать ошибку.

В первой итерации мы разделили синие и красные точки. И видим, что в левую часть попали тоже синие. Как указать следующей итерации, что эти точки ошибочные? Мы можем использовать в обучающей выборке веса. То есть с помощью весов мы можем указать, какие точки более важны для классификации, какие менее важны.

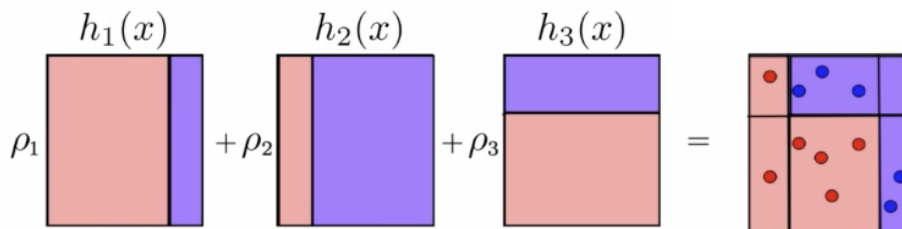
Для тех точек, которые классифицированы правильно, мы вес понижаем, которые неправильно – повышаем. Тогда:



Теперь при новом обучении классификатор обратит внимание на большие точки и постарается их отделить. Получим такую итоговую картину:

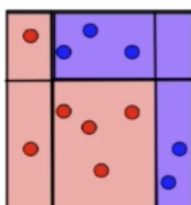


Все три классификатора мы строили последовательно. Объединим их с правильными весами. То есть каждый из этих классификаторов будет давать долю “уверенности” в правильном решении. Усреднив их между собой, получим достаточно точное решение:



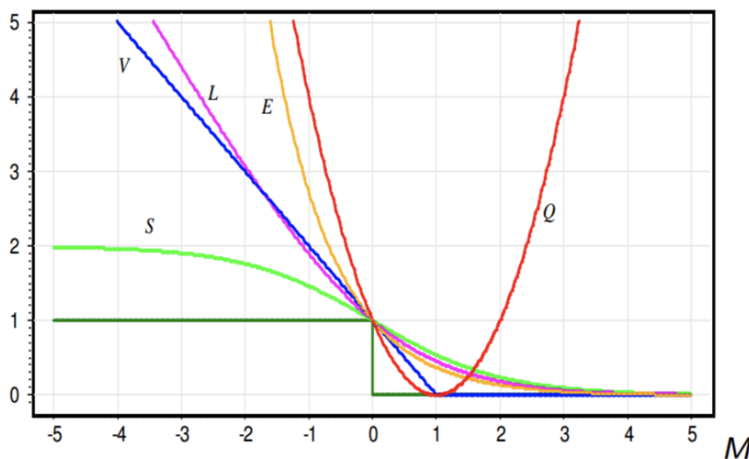
Здесь ρ_1 , ρ_2 , ρ_3 – веса, с которыми каждый классификатор вкладывается в итоговое решение.

Итоговая граница уже не прямая, а ломаная.

$$\hat{f}_T(x) = \sum_{t=1}^T \rho_t h_t(x) =$$


Таким образом, мы вышли из класса моделей по своей сложности. С помощью одного решающего дерева мы не могли нарисовать линейную разделяющую поверхность. Теперь мы можем строить ансамбль, повышая сложность гиперповерхности решения.

Рассмотрим картинку с различными функциями потерь в зависимости от отступа (margin):



$$\begin{aligned} Q(M) &= (1 - M)^2 \\ V(M) &= (1 - M)_+ \\ S(M) &= 2(1 + e^M)^{-1} \\ L(M) &= \log_2(1 + e^{-M}) \\ E(M) &= e^{-M} \end{aligned}$$

Обратим внимание на экспоненциальную функцию потерь:

$$E(M) = e^{-M}.$$

Чем больше модель ошибается, тем больше она получает функцию потерь.

Функция потерь от нашего ансамбля из T элементов:

$$\hat{f}_T(x) = \sum_{t=1}^T \rho_t h_t(x)$$

$$\begin{aligned} L(y_i, \hat{f}_T(x_i)) &= \exp(-y_i \hat{f}_T(x_i)) = \exp\left(-y_i \sum_{t=1}^T \rho_t h_t(x_i)\right) \\ &= \exp\left(-y_i \sum_{t=1}^{T-1} \rho_t h_t(x_i)\right) \cdot \exp(-y_i \rho_T h_T(x_i)) = w_i \cdot \exp(-y_i \rho_T h_T(x_i)) \end{aligned}$$

Получаем, что $\exp\left(-y_i \sum_{t=1}^{T-1} \rho_t h_t(x_i)\right) = \text{const}$ – функция потерь. Мы в качестве веса используем функцию потерь от предыдущего шага.

Такой метод называется **адаптивным бустингом (AdaBoost)**.

Адаптивный бустинг имеет недостатки:

- экспонента не очень хорошая функция, поскольку при определенных значениях она может быть очень большой.
- мы не всегда хотим очень сильно штрафовать модель за ошибки. В выбросах будет огромный отступ в неправильный класс, будет большая ошибка. Один отступ будет сильно перекашивать разделяющую поверхность.

2. Градиентный бустинг

Нейронные сети привлекали внимание людей достаточно давно. Первый всплеск интереса произошел в 40-х годах XX века, потом до конца века было еще несколько всплесков и падений. В 90-х годах произошло его падение по причине появления градиентного бустинга.

Нейронные сети в 90-х годах показывали достаточно качественные результаты, казалось, что они не переобучались. Но потом выяснилось, что просто за короткий промежуток накопилось много данных, но модели были небольшими, поэтому они не переобучались.

В 2000-м году был представлен градиентный бустинг. Он показывал ошеломительные результаты, был гораздо более эффективным, быстрым и простым.

Теория градиентного бустинга

Пусть у нас есть некоторая выборка $\{(x_i, y_i)\}_{i=1, \dots, n}$.

Пусть у нас задача регрессии. Хотя градиентный бустинг работает и на задаче классификации.

Функция потерь $L(y, f)$.

У нас есть оптимальная модель

$$\hat{f}(x) = \arg \min_{f(x)} L(y, f(x)) = \arg \min_{f(x)} \mathbb{E}_{x,y} [L(y, f(x))]$$

которую мы бы хотели найти.

Оптимальная модель – это та модель, которая доставляет нам минимум функции потерь.

Мы сужаем область поиска и параметризуем нашу модель:

$$\hat{f}(x) = f(x, \hat{\theta}),$$

то есть вместо поиска модели мы будем искать оптимальные параметры модели.

То есть мы можем переформулировать задачу оптимизации следующим образом:

$$\hat{\theta} = \arg \min_{\theta} \mathbb{E}_{x,y}[L(y, f(x, \theta))]$$

Полученная задача имеет отношение к постановке любой задачи обучения с учителем.

Рассмотрим ограничения, накладываемые на модель и на используемую функцию потерь.

Пусть наша модель будет ансамблем из $(t - 1)$ элемента.

$$\hat{f}(x) = \sum_{i=0}^{t-1} \hat{f}_i(x),$$

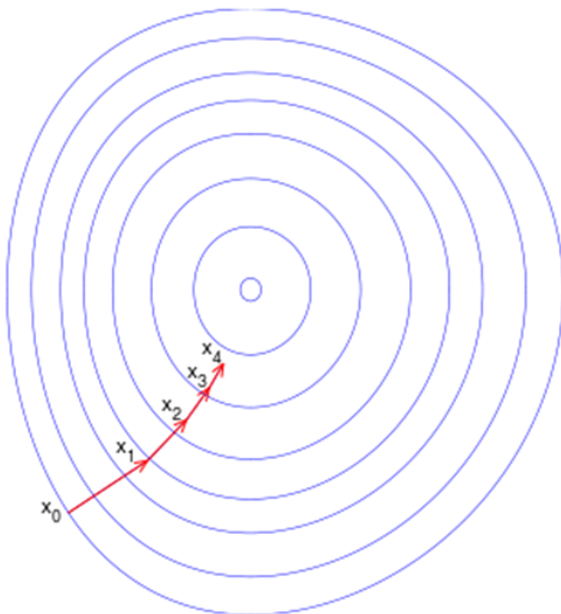
Перейдем к шагу t . Будем обучать f_t модель на основе предыдущих. Стоит обратить внимание, что внутри функций f_i спрятаны $\rho_i h_i$.

Мы хотим на шаге t найти оптимальный вес ρ_t и параметр θ_t , как решение задачи минимизации:

$$(\rho_t, \theta_t) = \arg \min_{\rho, \theta} \mathbb{E}_{x,y}[L(y, \hat{f}(x) + \rho \cdot h(x, \theta))],$$

$$\hat{f}_t(x) = \rho_t \cdot h(x, \theta_t)$$

Для минимизации мы можем взять градиентный спуск.



Градиентным шагом будет не изменение вектора параметров, а новая модель в нашем ансамбле. Каждая модель будет приближать нас к оптимуму, если мы будем двигаться вдоль антиградиента нашей функции потерь.

Предположим, у нас есть какая-то начальная ситуация, где наша модель вообще ничего “не знает”. Например, это просто константа. У нас есть некоторая функция потерь x_0 (см. рисунок). Мы берем первый элемент ансамбля, обучаемся на минимизации ошибки и попадаем в новую точку x_1 . Теперь мы хотим понизить ошибку текущего ансамбля, то есть переместиться в x_2 . Как понять, куда переместиться в пространство функций, если по пространству функции дифференцировать не можем (у нас решающее дерево)?

У нас есть наш ансамбль:

$$\hat{f}(x) = \sum_{i=0}^{t-1} \hat{f}_i(x),$$

Мы можем посчитать градиент нашей функции потерь по предсказаниям модели – частные производные:

$$r_{it} = - \left[\frac{\partial L(y_i, f(x_i))}{\partial f(x_i)} \right]_{f(x)=\hat{f}(x)}, \quad i = 1, \dots, n,$$

У нас есть ограничения на структуру модели и на функцию потерь. Мы потребуем, чтоб наша функция потерь была дифференцируема по своим аргументам. У нее два аргумента: истинное значение целевой переменной и предсказанное значение целевой переменной. Именно на предсказанное значение целевой переменной мы и будем обращать внимание.

Частная производная по предсказаниям модели – что-то непонятное. Можно провести параметризацию. Величину r_{it} мы посчитаем для каждой точки нашей обучающей выборки. Для каждой точки мы можем оценить антиградиент функции потерь по предсказаниям модели.

Теперь вектор градиента мы можем использовать, как новое значение целевой переменной на шаге t .

$$\theta_t = \arg \min_{\theta} \sum_{i=1}^n (r_{it} - h(x_i, \theta))^2,$$

Теперь мы можем понять, каким образом нам нужно изменить предсказание нашей модели на каждой из точек, чтобы модель получила наименьшую ошибку. Затем решаем задачу аппроксимации антиградиента с помощью новой модели на шаге t . Это задача регрессии, мы ее умеем решать.

Теперь остается вопрос: где взять ρ_t ?

$$\rho_t = \arg \min_{\rho} \sum_{i=1}^n L(y_i, \hat{f}(x_i) + \rho \cdot h(x_i, \theta_t))$$

Исходная функция потерь дифференцируема, её можно минимизировать.

В результате, мы построили ансамбль типа градиентный бустинг.

Градиентный бустинг является наиболее подходящим к огромному числу задач с табличными данными. Нейронные сети хорошо работают с изображениями, звуками, текстами, графами. А с таблицами работает градиентный бустинг сопоставимо, а зачастую гораздо лучше.

Задача линейной регрессии

Рассмотрим задачу линейной регрессии со среднеквадратичной функцией ошибки MSE:

$$r_{it} = - \left[\frac{\partial L(y_i, f(x_i))}{\partial f(x_i)} \right]_{f(x)=\hat{f}(x)} = -2(\hat{y}_i - y_i) \propto \hat{y}_i - y_i$$

В этой задаче градиентный бустинг приобретает лаконичную и простую форму. Каждая модель учится минимизировать ошибку предыдущей напрямую. Каждая следующая модель учится на регрессионных остатках предыдущей модели.

NB. Адабуст тоже является частным случаем градиентного бустинга с подходящей функцией потерь.

3. Визуализация градиентного бустинга

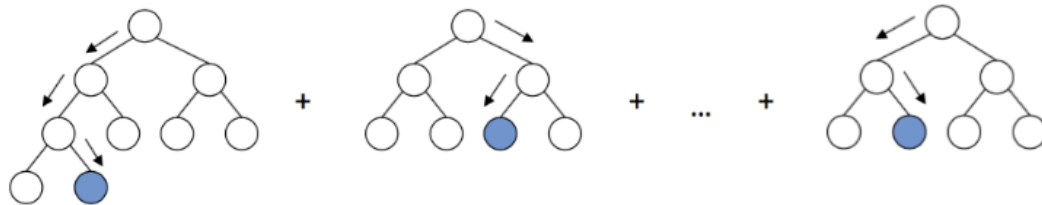
Визуализацию градиентного бустинга можно внимательно изучить по [ссылке](#) самостоятельно. Она интерактивна, можно самим задавать нужные параметры.

Мы посмотрим на то, как строится решающее дерево в задаче регрессии, и как строится алгоритм бустинга над решающими деревьями, конкретно – градиентного бустинга в задаче регрессии.

Решающее дерево

Gradient Boosting explained [demonstration]

Jun 24, 2016 • Alex Rogozhnikov •

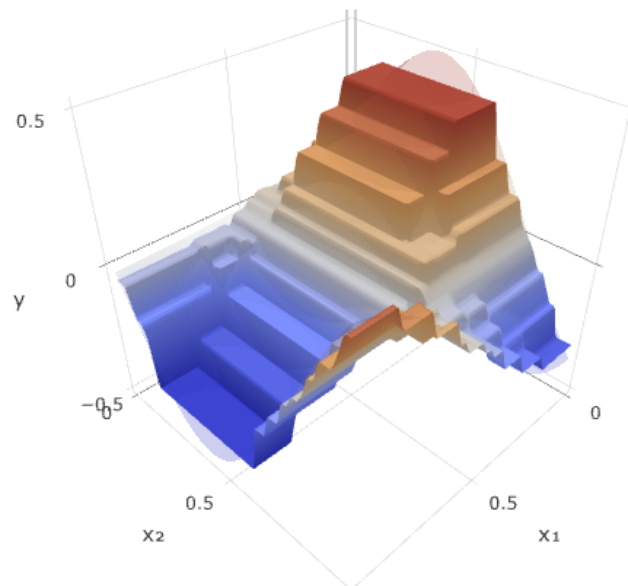


Gradient boosting (GB) is a machine learning algorithm developed in the late '90s that is still very popular. It produces state-of-the-art results for many commercial (and academic) applications.

This page explains how the gradient boosting algorithm works using several interactive visualizations.

Decision Tree Visualized

semi-transparent target function $f(\mathbf{x})$ and tree prediction $d_{\text{tree}}(\mathbf{x})$



Tree depth: 6

Look from above

We take a 2-dimensional regression problem and investigate how a tree is able to reconstruct the function $y = f(\mathbf{x}) = f(x_1, x_2)$. Play with the tree depth, then look at the tree-building process from above!

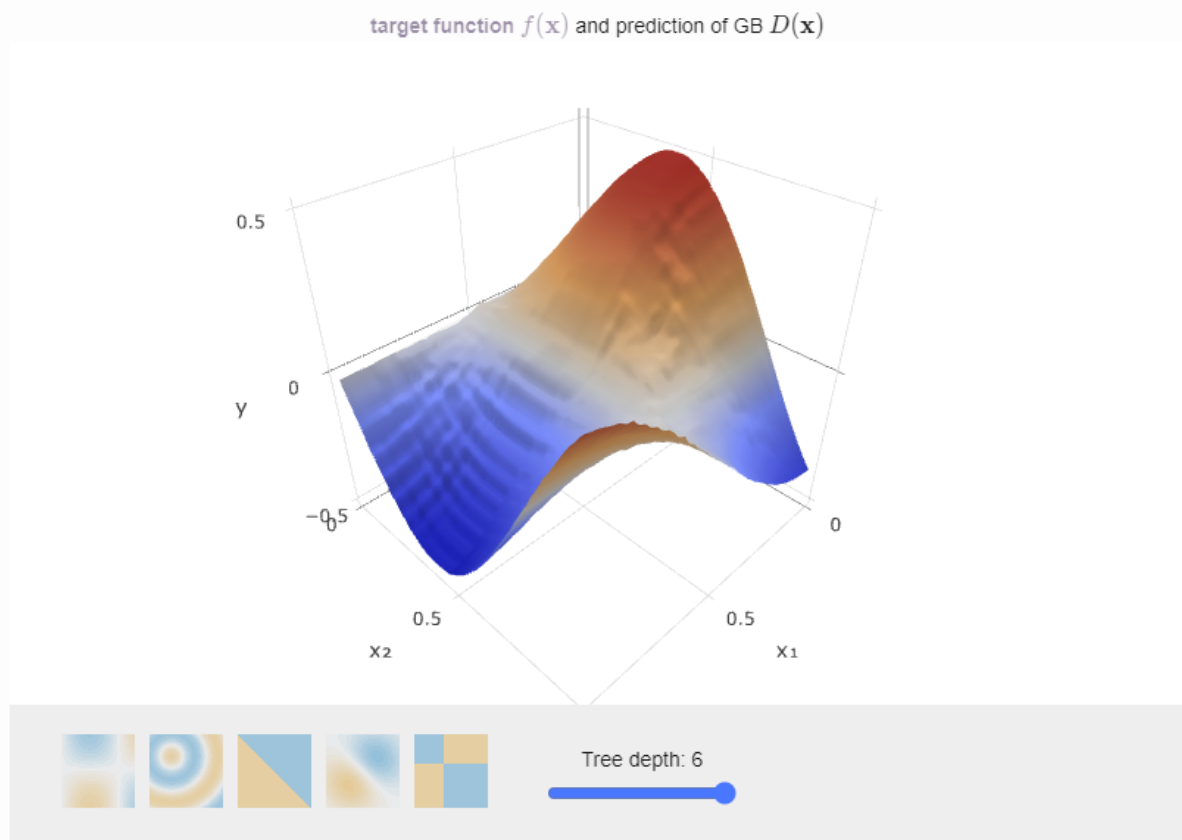
Решающее дерево глубины 0 – это просто константа. Константа конечно сложную поверхность не может правильно описать. Дерево глубины – решающий пенн, получается ступенька. При увеличении глубины качество аппроксимации улучшается. Тем не менее аппроксимация все равно остается равной и ломаной.

Градиентный бустинг

Перейдем к визуализации градиентного бустинга.

Gradient Boosting Visualized

This demo shows the result of combining 100 decision trees.



Not bad, right? As we see, gradient boosting is able to provide smooth detailed predictions by combining many trees of very limited depth (cf. with the single decision tree above!).

Опять же, ансамбль из 100 деревьев глубины 0 (100 констант) не дает хорошего результата. При увеличении глубины поверхность аппроксимации становится более гладкой, и она все ближе к той поверхности, которую мы хотим описать, по сравнению с одним деревом той же глубины.

Итак, градиентный бустинг позволяет описывать гораздо более сложные зависимости по сравнению с одним решающим деревом, используя при этом простые модели в ансамбле.

Важно! Градиентный бустинг очень легко переобучается. Мы явным образом эксплуатируем возможность модели подстроиться под антиградиент функции потерь. Функцию потерь мы считаем на обучающей выборке, поэтому мы по сути ищем наилучший способ подстройки под обучающую выборку, то есть наилучший способ переобучения. Надо быть очень осторожными при построении ансамбля типа бустинг. Нужно следить за качеством модели на валидации и строить валидационные пайплайны так, чтобы мы могли им доверять.

4. CatBoost

Градиентный бустинг по праву снискал славу во многих задачах и в первую очередь в задачах, где необходимо обрабатывать табличные данные. Стоит упомянуть алгоритм и библиотеку CatBoost.

CatBoost – одна из крупнейших библиотек на текущий момент, которые используются в построении алгоритмов типа градиентный бустинг. Вместе с ним можно выделить XBoost, который поддерживается академическим сообществом. Эти библиотеки показывают отличные результаты на многих прикладных задачах и соревнованиях.

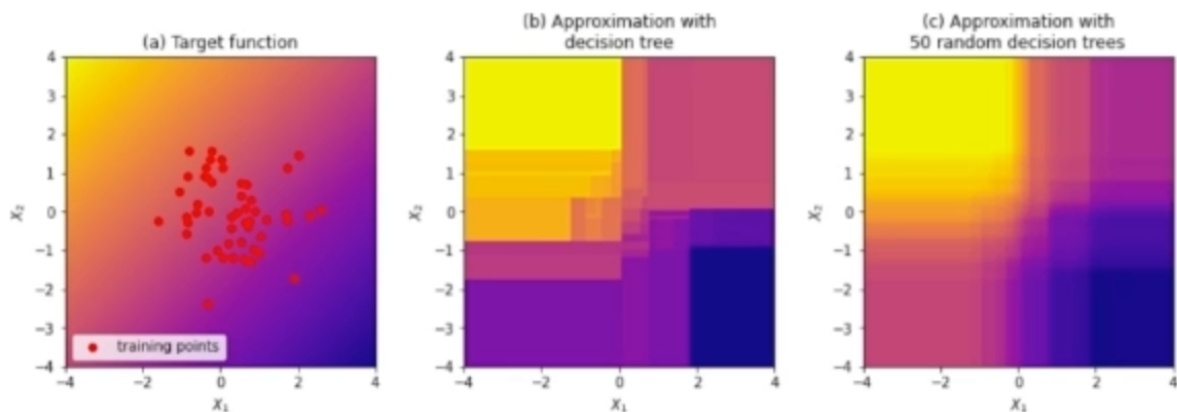
CatBoost выделяется несколькими свойствами:

1. В нем используются специальные механизмы для обработки категориальных признаков (отсюда название)
2. CatBoost имеет несколько вариантов построения деревьев.

В основе также лежат решающие деревья, но бустинг можно строить и над логистическими регрессиями, и другими нелинейными моделями.

Стоит выделить основную идею CatBoost: CatBoost во многом опирается на то, что модель должна быстро работать. А значит она не должна иметь сложных ветвлений. Желательно, чтобы деревья качественно представлялись в памяти компьютера.

Рассмотрим пример из [статьи](#):

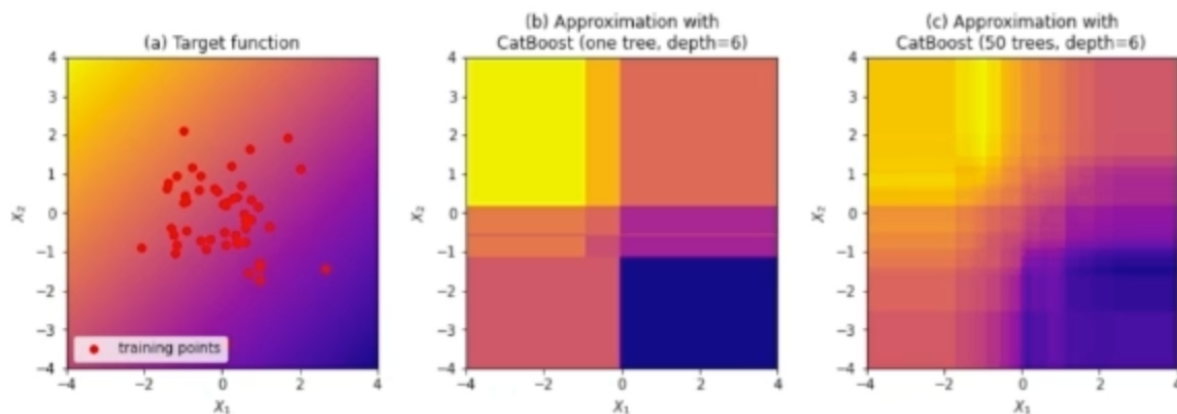


На картинке есть несколько красных обучающих точек для зависимости $y = x_1 + x_2$.

Видим, что одно обучающее дерево плохо экстраполирует предсказание.

И есть результат для 50 случайных деревьев. Экстраполяция также не очень хорошо работает.

Применили к задаче алгоритм CatBoost:



Картинки получились достаточно похожие. Механизм экстраполяции чуть лучше у CatBoost, но стоит обратить внимание, что CatBoost умеет строить решающие деревья таким образом, что на каждом уровне, то есть на одинаковом расстоянии от корня:

- используется один и тот же признак для разбиения. Разбиение происходит не по оптимальному признаку. Здесь оптимальный признак выбирается для всего уровня.
- в CatBoost используется один и тот же порог для разбиения в каждой из вершин.

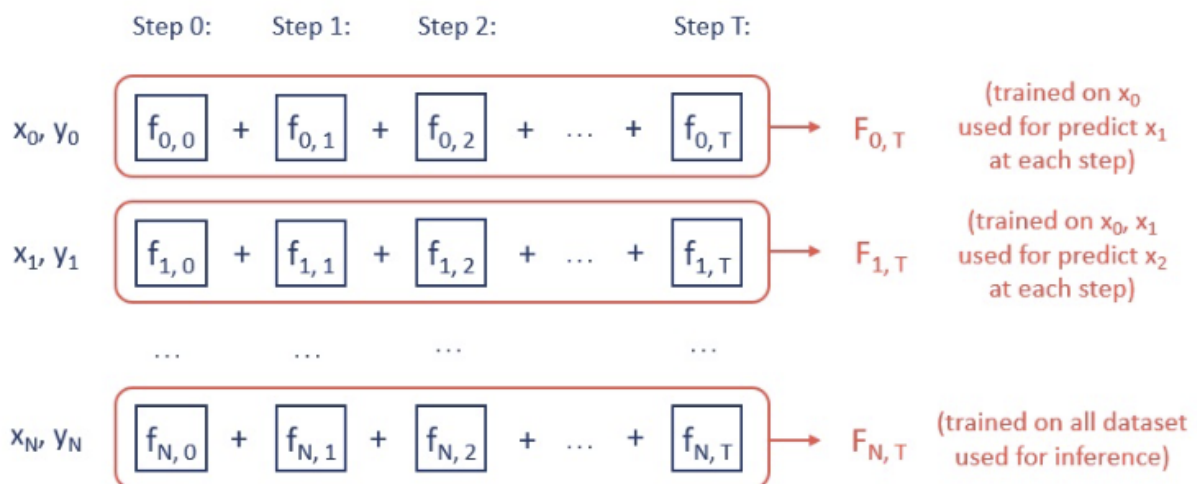
Это приводит к тому, что дерево гораздо проще записать, и по сути его можно представить в виде некоторой решающей таблицы, где каждому столбцу будет соответствовать какое-то значение.

Это позволяет CatBoost работать очень быстро.

В CatBoost используется идея **упорядоченного бустинга**.

Вспомним, как работает бустинг. Каждый элемент ансамбля (например, каждое дерево) обучается на всей обучающей выборке. Потом делается предсказание на всей обучающей выборке, считается ошибка на всей обучающей выборке, считается антиградиент функции потерь, и следующий элемент ансамбля аппроксимирует этот антиградиент. Это не очень эффективно, потому что мы считаем антиградиент, который является таргетом, на той же обучающей выборке, на которой и происходило обучение. Оценка получается смещенной.

В упорядоченном бустинге предлагается упорядочить каким-то образом все объекты, и теперь обучать N моделей для N объектов.



То есть для каждого объекта происходит предсказание от дерева, которое его раньше “не видело”. Но при этом на каждом шаге построения бустинга строить N деревьев, где N – размер выборки. Выборка может исчисляться миллионами объектов. Это также не очень эффективно.

В CatBoost применяется чуть другой подход, где на каждом шаге по сути есть возможность работать даже с одним деревом.

```

input :  $\{(x_i, y_i)\}_{i=1}^n, I, \alpha, L, s$ 
1   $\sigma_r \leftarrow$  random permutation of  $[1, n]$  for  $r = 0 \dots s$ ;
2   $M_0(i) \leftarrow 0$  for  $i = 1 \dots n$ ;
3  for  $j \leftarrow 1$  to  $\lceil \log_2 n \rceil$  do
4     $M_{r,j}(i) \leftarrow 0$  for  $r = 1 \dots s, i = 1 \dots 2^{j+1}$ ;
5  for  $t \leftarrow 1$  to  $I$  do
6     $T_t, \{M_r\}_{r=1}^s \leftarrow \text{BuildTree}(\{M_r\}_{r=1}^s, \{(x_i, y_i)\}_{i=1}^n, \alpha, L, \{\sigma_i\}_{i=1}^s)$ ;
7     $\text{leaf}_0(i) \leftarrow \text{GetLeaf}(x_i, T_t, \sigma_0)$  for  $i = 1 \dots n$ ;
8     $\text{grad}_0 \leftarrow \text{CalcGradient}(L, M_0, y)$ ;
9    foreach  $\text{leaf } j$  in  $T_t$  do
10      $b_j^t \leftarrow -\text{avg}(\text{grad}_0(i) \text{ for } i : \text{leaf}_0(i) = j)$ ;
11    for  $i = 1 \dots n$  do
12      $M_0(i) \leftarrow M_0(i) + \alpha b_{\text{leaf}_0(i)}^t$ 
13 return  $F(x) = \sum_{t=1}^I \sum_j \alpha b_j^t \mathbb{1}_{\text{GetLeaf}(x, T_t, \text{ApplyMode})=j}$ ;

```

CatBoost позволяет неплохо обрабатывать категориальные признаки.

Категориальные признаки, как мы знаем, не упорядочены (красный, синий, зеленый).

Но когда категориальных признаков мало, их можно заменить one-hot векторами.

Тогда красному будет соответствовать (1, 0, 0), зеленому – (0, 1, 0), синему – (0, 0, 1).

Для 100 000 признаков это делать неудобно. CatBoost упорядочивает данные

случайным образом. И согласно этой упорядоченности происходит таргет

кодирование, то есть замена категориального признака на статистику целевой

переменной. Для подсчета этой статистики используется лишь подвыборка из

элементов, чей индекс меньше или равен текущему объекту. Для объектов в начале

это работает не очень хорошо, поскольку для них маленькая выборка. Поэтому это

делается несколько раз для различных случайных упорядочиваний. И тогда вместо

категориальных признаков появляются числовые, и с этим можно работать.

Дополнительные материалы для самостоятельного изучения

1. http://arogozhnikov.github.io/2016/06/24/gradient_boosting_explained.html
2. <https://habr.com/ru/company/ods/blog/645887/>