Градиентный бустинг

Цель занятия

В результате обучения на этой неделе:

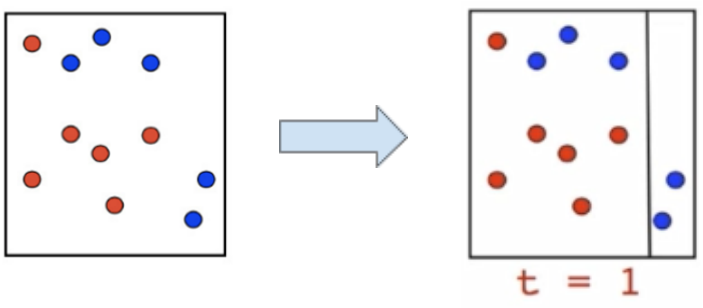
- вы погрузитесь в изучение градиентного бустинга;
- познакомитесь с алгоритмом AdaBoost;
- рассмотрите визуализацию градиентного бустинга и посмотрите, как строится алгоритм градиентного бустинга над решающими деревьями;
- научитесь решать задачи классификации и регрессии с помощью градиентного бустинга;
- познакомитесь с библиотекой CatBoost;
- познакомитесь на практике с библиотеками xgboost и CatBoost;
- разберете примеры использования градиентного бустинга.

План занятия

Конспект занятия

1. Бустинг

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



Логично, что с помощью одного решающего дерева глубины 1 мы эту задачу никак не решим. Однако, используя бустинг, мы решим эту задачу с помощью решающих пней в качестве базовых алгоритмов.

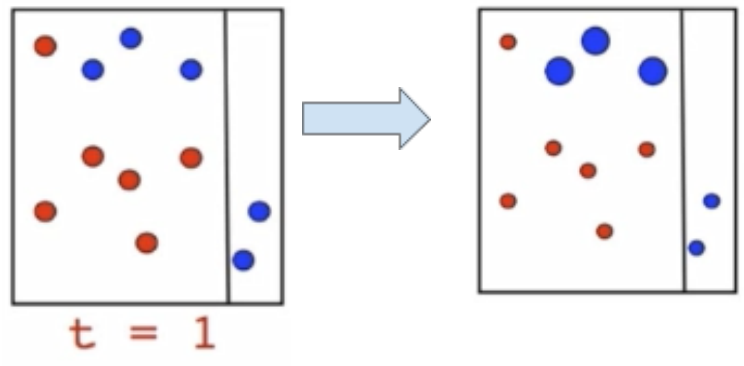
Простое усреднение в данном методе использовать нельзя. Усреднение — линейная комбинация, решающий пень — линейная модель. Линейная комбинация линейных моделей ни к чему не приведет.

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.

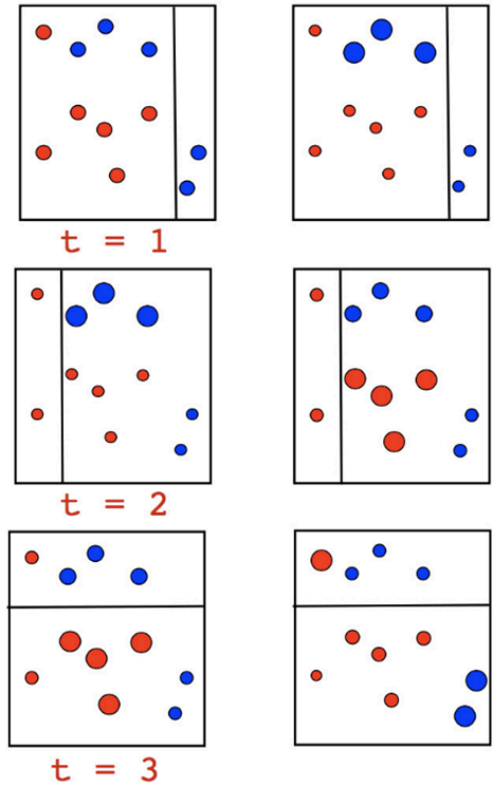
Для тех точек, которые классифицированы правильно, мы вес понижаем, которые неправильно — повышаем.

Тогда:



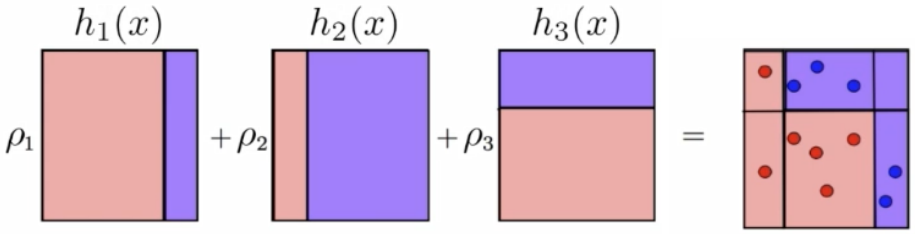
Теперь при новом обучении классификатор обратит внимание на большие точки и постарается их отделить.

Получим такую итоговую картину:



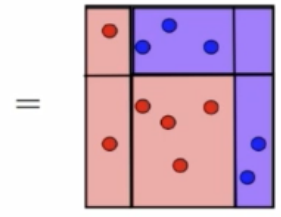
Все три классификатора мы строили последовательно. Объединим их с правильными весами. То есть каждый из этих классификаторов будет давать долю «уверенности» в правильном решении.

Усреднив их между собой, получим достаточно точное решение:



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

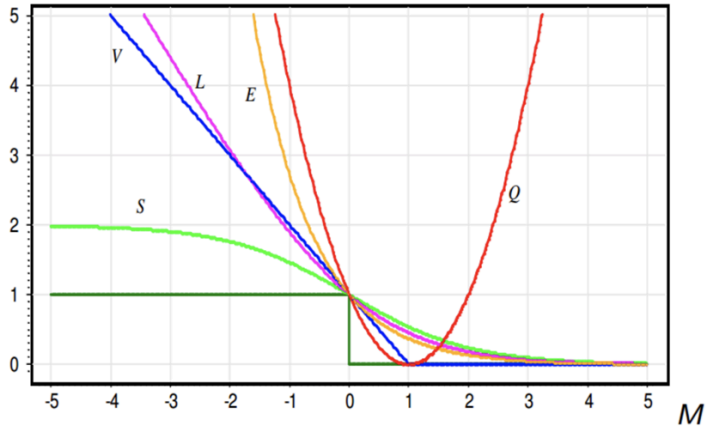
Итоговая граница уже не прямая, а ломаная.



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

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

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



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

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:

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

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

Получаем, что $exp\left(-y_i \sum_{t=1}^{T-1} \rho_t h_t(x_i)\right) = const$ – 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 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 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, которую мы бы хотели найти:

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

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.

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.

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

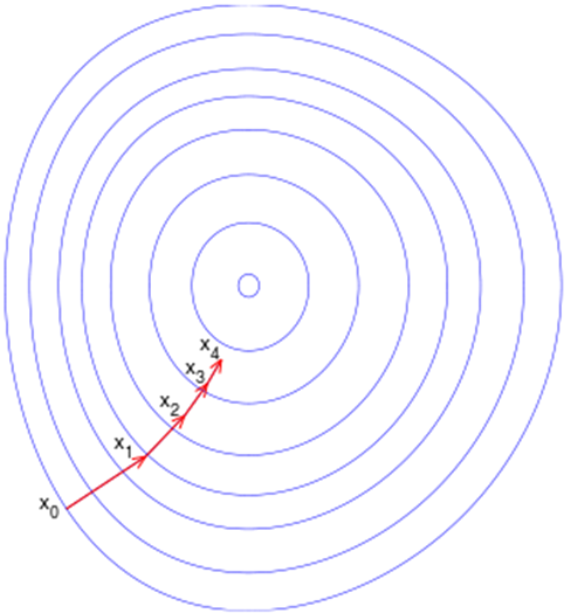
Перейдем к шагу t. Будем обучать f_t модель на основе предыдущих. Стоит обратить внимание, что внутри функций f_i спрятаны $\rho_i h_i$.

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:

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

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, ее можно минимизировать.

В результате мы построили ансамбль типа градиентный бустинг.

Градиентный бустинг — наиболее подходящий 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.

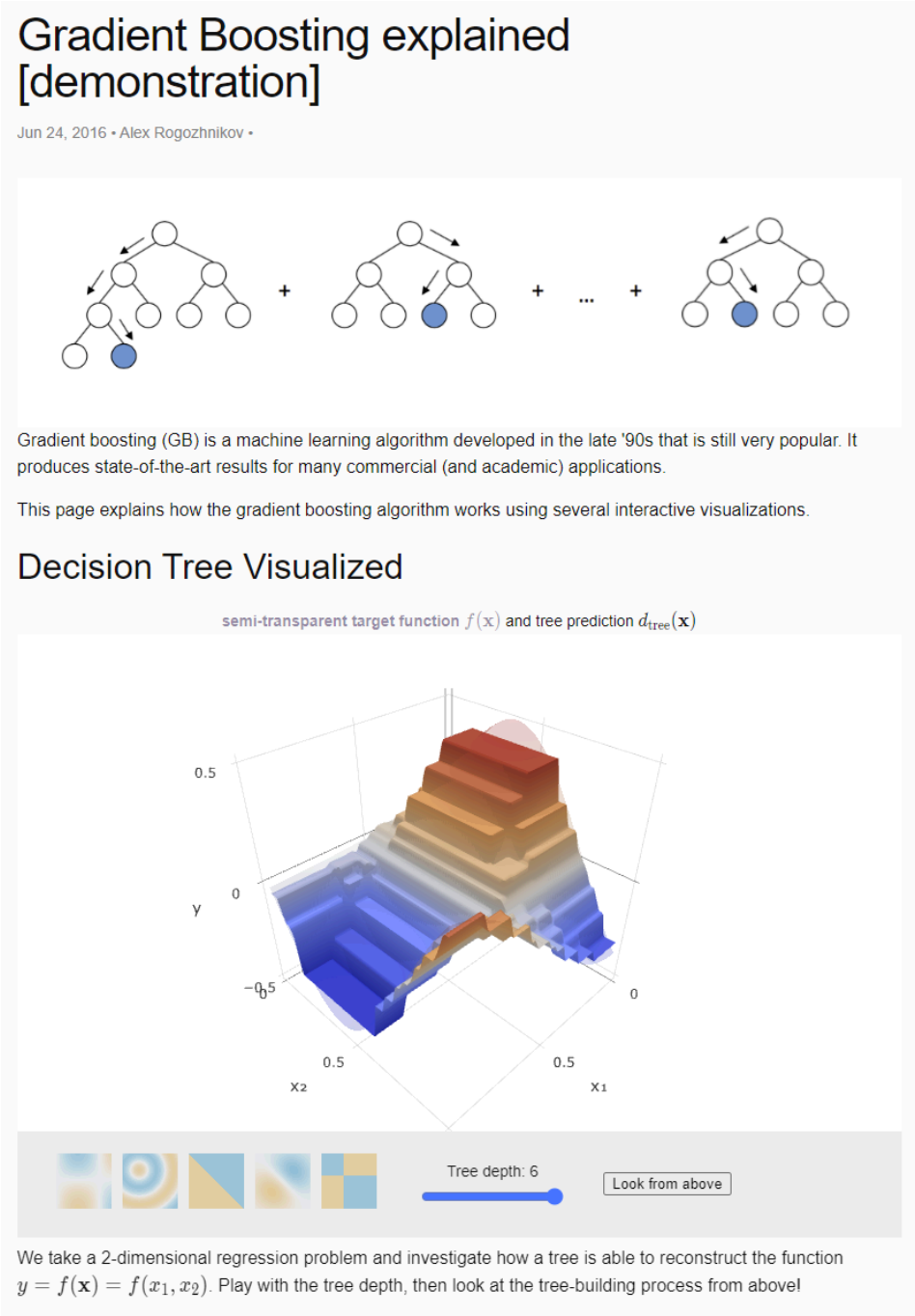
NB. AdaBoost тоже является частным случаем градиентного бустинга с подходящей функцией потерь.

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.

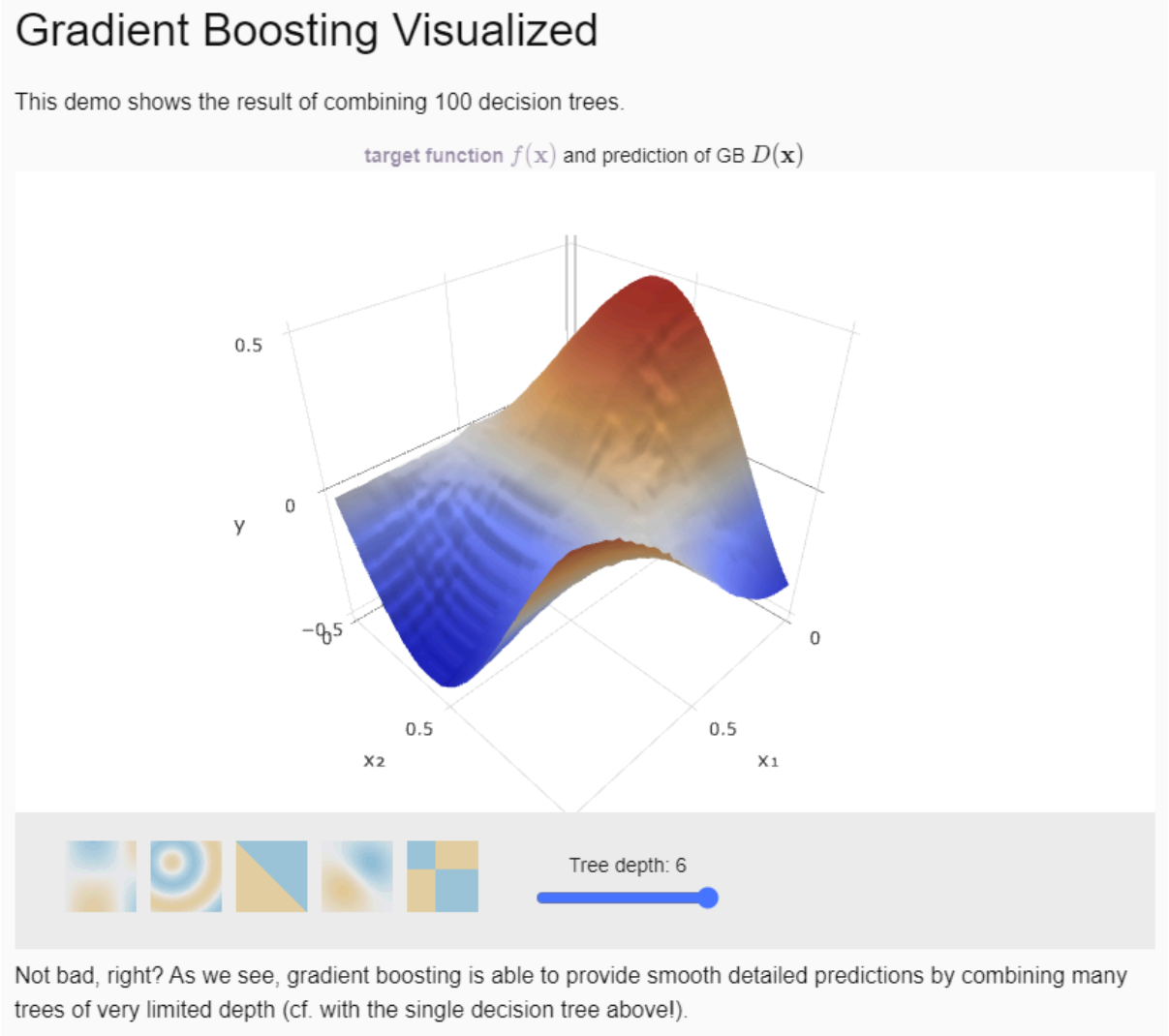
Решающее дерево



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

Градиентный бустинг

Перейдем к визуализации градиентного бустинга.



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

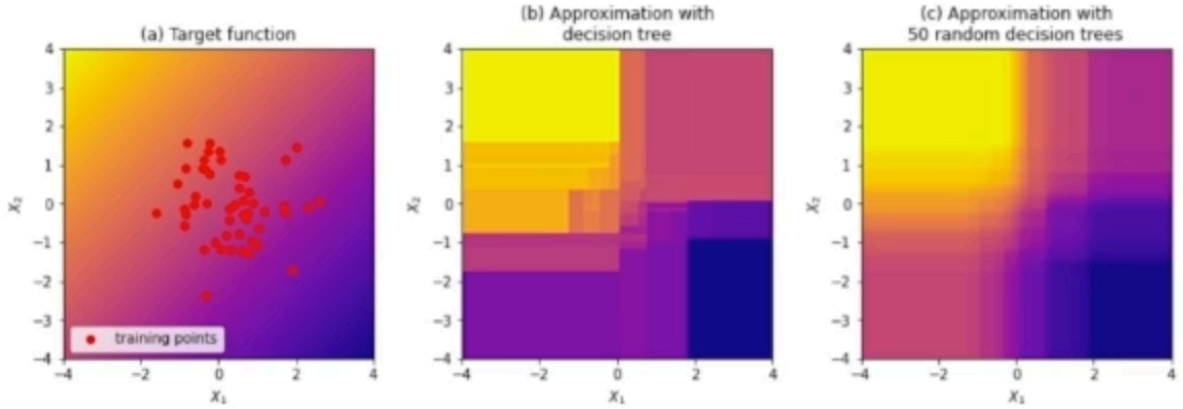
CatBoost выделяется несколькими свойствами:

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

В основе 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.

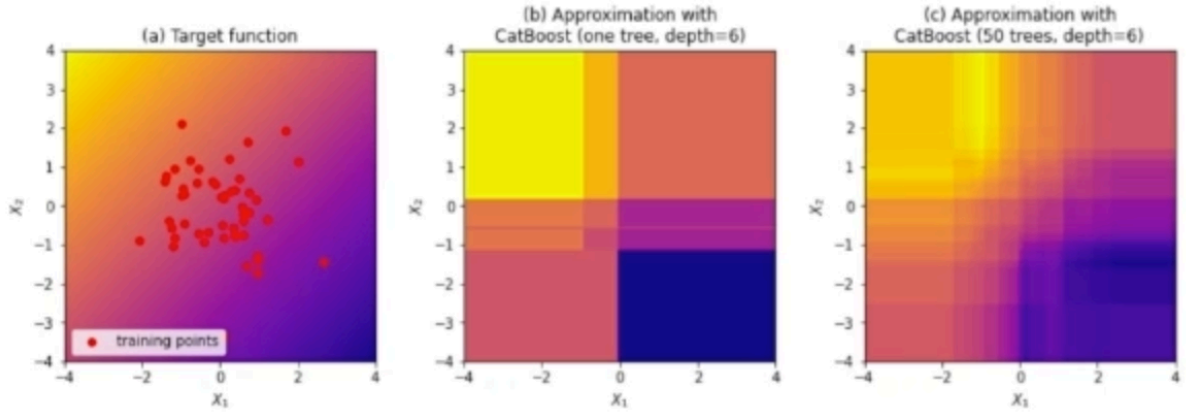
Пример. Рассмотрим пример из статьи:



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. 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.

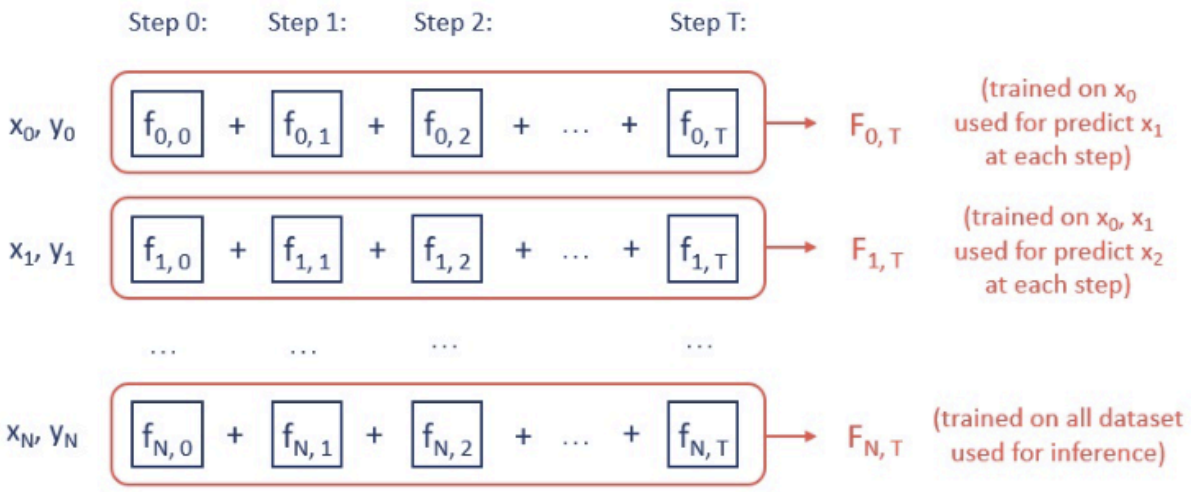
Это приводит к тому, что дерево гораздо проще записать и можно представить в виде некоторой решающей таблицы, где каждому столбцу будет соответствовать какое-то значение.

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 есть возможность работать даже с одним деревом.

input : $\{(x_i, y_i)\}_{i=1}^{n}, I, \alpha, L, s$

$\sigma_r \leftarrow$ random permutation of $[1, n]$ for $r = 0 \dots s$;

$M_0(i) \leftarrow 0$ for $i = 1 \dots n$;

for $j \leftarrow 1$ **to** $\lceil \log_2 n \rceil$ **do**

  $M_{r,j}(i) \leftarrow 0$ for $r = 1 \dots s,\ i = 1 \dots 2^{j+1}$;

for $t \leftarrow 1$ **to** I **do**

  $T_t, \{M_r\}_{r=1}^{s} \leftarrow BuildTree(\{M_r\}_{r=1}^{s}, \{(x_i, y_i)\}_{i=1}^{n}, \alpha, L, \{\sigma_i\}_{i=1}^{s})$;

  $leaf_0(i) \leftarrow GetLeaf(x_i, T_t, \sigma_0)$ for $i = 1 \dots n$;

  $grad_0 \leftarrow CalcGradient(L, M_0, y)$;

  foreach $leaf\, j$ **in** T_t **do**

    $b_j^t \leftarrow -\text{avg}(grad_0(i)$ for $i : leaf_0(i) = j)$;

  for $i = 1 \dots n$ **do**

    $M_0(i) \leftarrow M_0(i) + \alpha b^t_{leaf_0(i)}$

return $F(x) = \sum_{t=1}^{I} \sum_j \alpha b_j^t \mathbb{1}_{GetLeaf(x, T_t, ApplyMode)=j}$;

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 кодирования, 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.

5. Сравнение градиентного бустинга с другими ансамблевыми методами

Рассмотрим на конкретном примере, как построить бустинг по уже имеющимся данным, и сравним результаты с методами случайного леса и логистической регрессии.

У нас имеются:

- данные;
- дифференцируемая функция потерь, которую мы хотим минимизировать;
- семейство алгоритмов, из которых мы будем выбирать нашу базовую модель.

В бустинге базовые модели могут быть абсолютно из разных семейств. На первом шаге мы можем использовать логистическую регрессию, на втором — kNN, на третьем — SVM, на четвертом — решающее дерево. Как правило, так не делают, и чаще всего бустинг строят над решающими деревьями, поскольку решающие деревья не сильно требовательны к предобработке данных.

Зафиксируем число итераций M (или будем его адаптивно фиксировать в процессе обучения).

Пусть у нас есть следующие **данные**:

$$y = cos(x) + \epsilon,\ \epsilon \sim N\left(0,\ \frac{1}{5}\right),\ x \in [-5,\ 5]$$

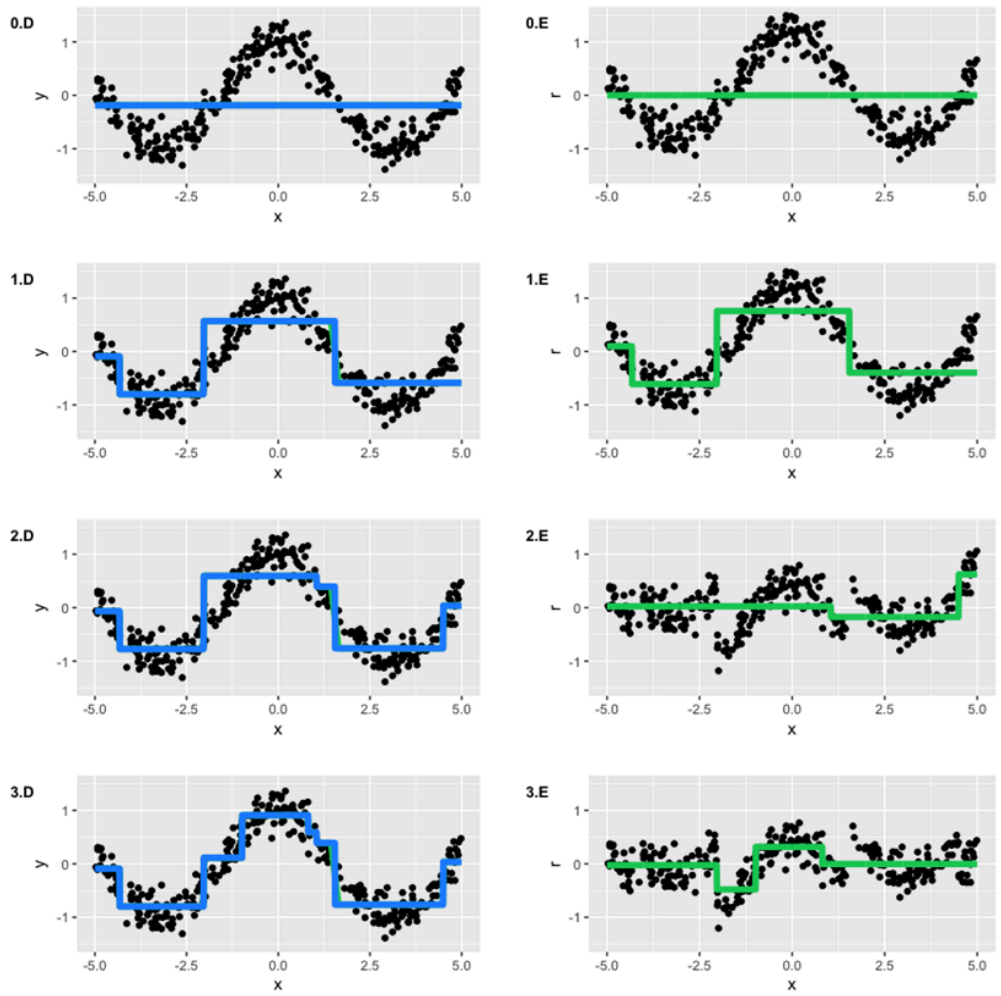
Функция потерь: MSE.

Семейство алгоритмов: решающие деревья глубины 2.

Число итераций: 3, будем строить последовательно 3 дерева.

Инициализация: константа — среднее.

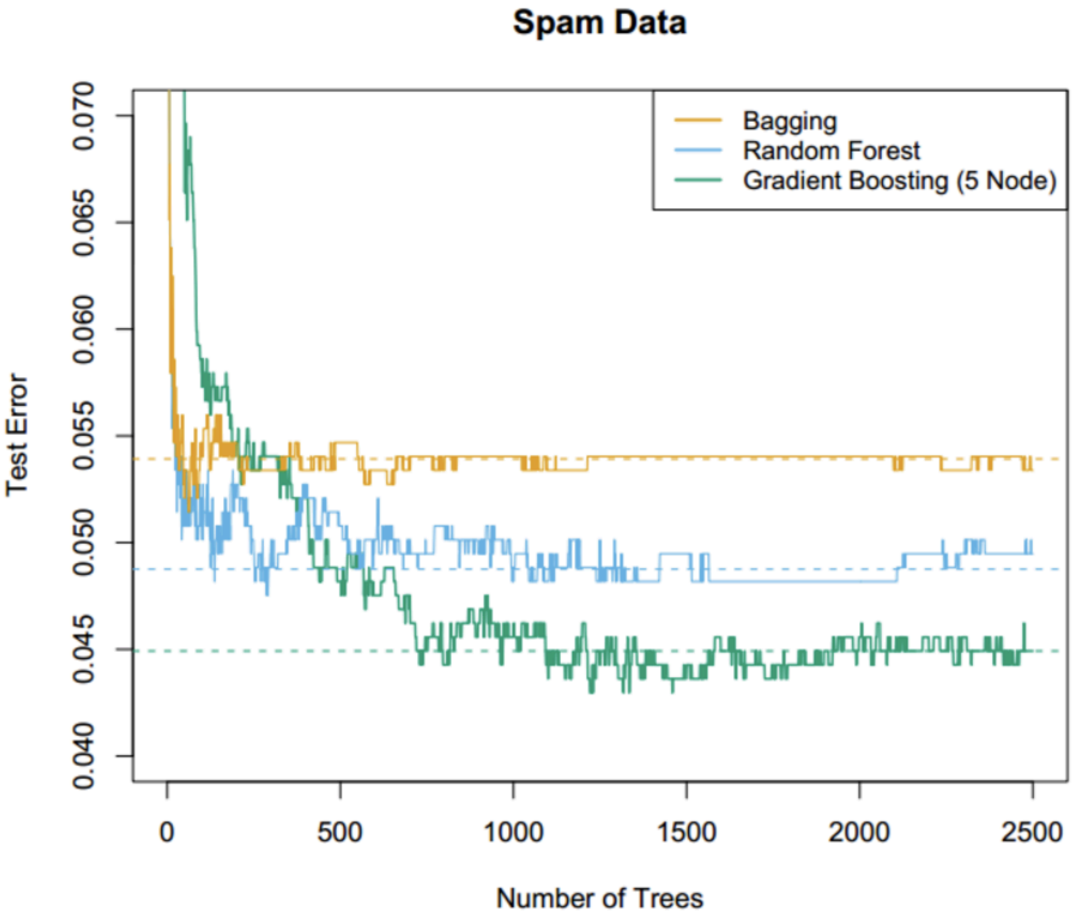
Получаем следующие результаты на графике:



Графики слева — то, что мы имеем на текущий момент нашей итерации, справа — то, что добавили с помощью нового элемента ансамбля. На последнем шаге справа видим, что сигнал становится все больше похож на белый шум. Как только он полностью станет белым шумом, это будет означать, что мы «вытащили» из наших данных все, что нужно. Но многомерные данные не очень удобно проверять, конечно.

На последнем шаге слева видим итоговый ансамбль. Мы описали косинус с помощью деревьев глубины 2.

Теперь сравним градиентный бустинг на практических задачах с уже знакомыми нам алгоритмами:

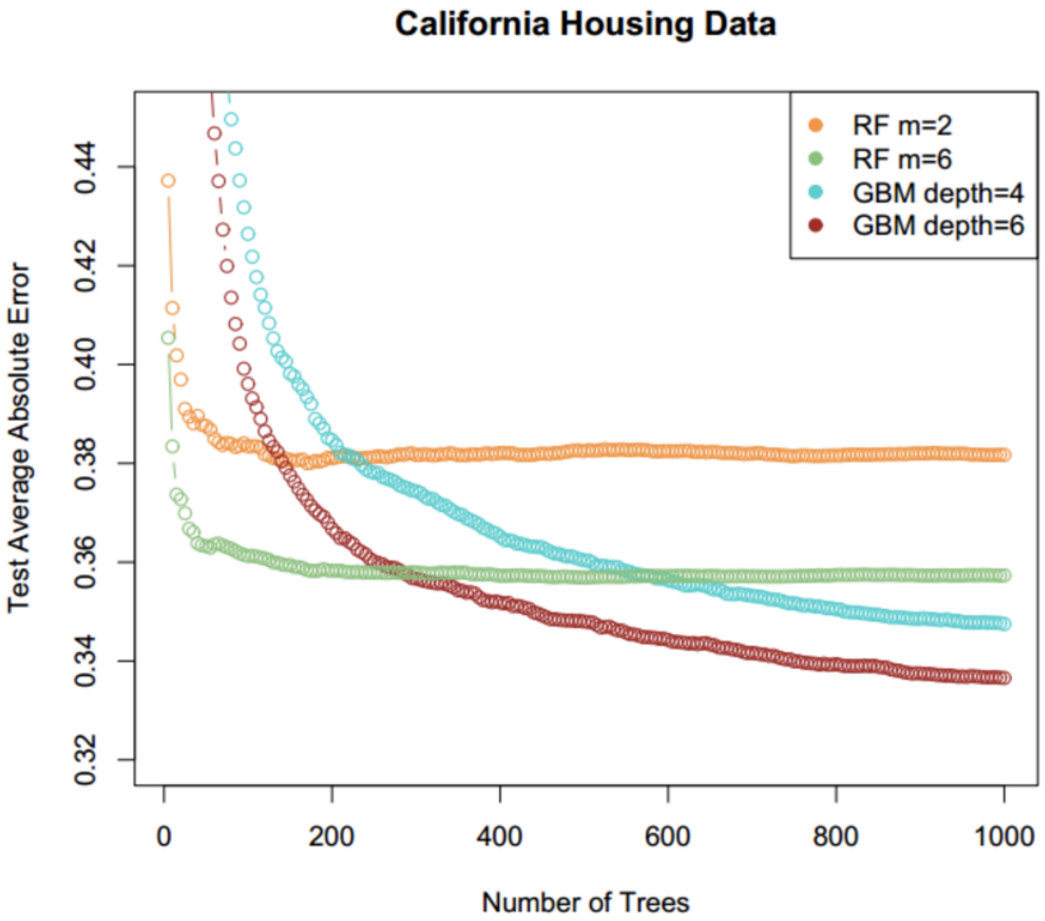


Видим, что в зависимости от числа деревьев ошибка падает у бэггинга и случайного леса сначала быстрее. Это из-за того, что у бустинга деревья в ансамбле неглубокие. Примерно на 50 деревьях бэггинг выходит на насыщение, и их качество практически не меняется. Случайный лес выходит на насыщение чуть позже.

При большом количестве деревьев появляются похожие (скоррелированные) деревья, поэтому ошибки становятся скоррелированными, и больше не уменьшаются.

Градиентный бустинг строит деревья последовательно в отличие от бэггинга и случайного леса. Каждое следующее дерево улучшает качество. И бустинг на графике выходит на еще более низкое плато функции ошибки.

Еще один пример применения деревьев разной глубины:



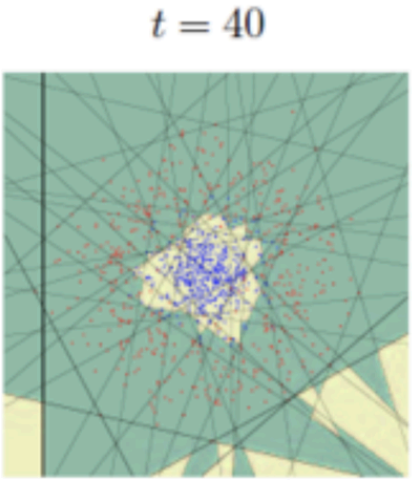
Здесь сравниваются градиентный бустинг и случайный лес с ансамблем деревьев глубины 4 и 6.

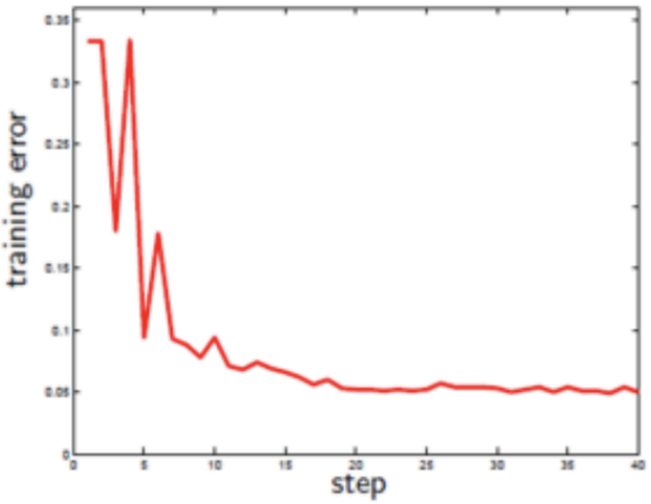
Как видим, градиентный бустинг лучше минимизирует ошибку, но требует гораздо более тонкой настройки.

Вопрос: что если строить ансамбль типа бустинг над линейными моделями? Например, над линейными регрессиями.

Ответ: нет. Ансамбль типа бустинг — это линейная сумма, взвешенная сумма всех используемых линейных моделей. Линейная комбинация линейных моделей — линейная модель. Она не выведет нас из класса линейных моделей, и более подходящую зависимость под данную задачу мы не получим.

Но рассмотрим тогда такую классическую задачу:





Здесь нелинейная разделяющая поверхность, двумерный случай. Справа представлена визуализация работы бустинга. Каждый линейный классификатор дает нам некоторую разделяющую поверхность.

Каким образом в данной задаче градиентный бустинг минимизировал ошибку?

Дело в том, что это логистическая регрессия, то есть бинарная классификация, — сигмоида от линейного преобразования. Это нелинейная модель. Она описывает линейную разделяющую поверхность, но само отображение является нелинейным. Поэтому линейная комбинация нелинейных моделей никаких ограничений не имеет.

На практике логистическая регрессия в составе бустинга встречается нечасто. Преимуществом такой модели является то, что она очень быстрая.

Сравним градиентный бустинг с методом случайного леса не только с точки зрения качества, но и еще с точки зрения скорости.

Случайный лес	Бустинг
Параллелизуется. Работает очень быстро	Не параллелизуется на уровне отдельных элементов ансамбля. Каждый новый элемент строится после построения предыдущего.

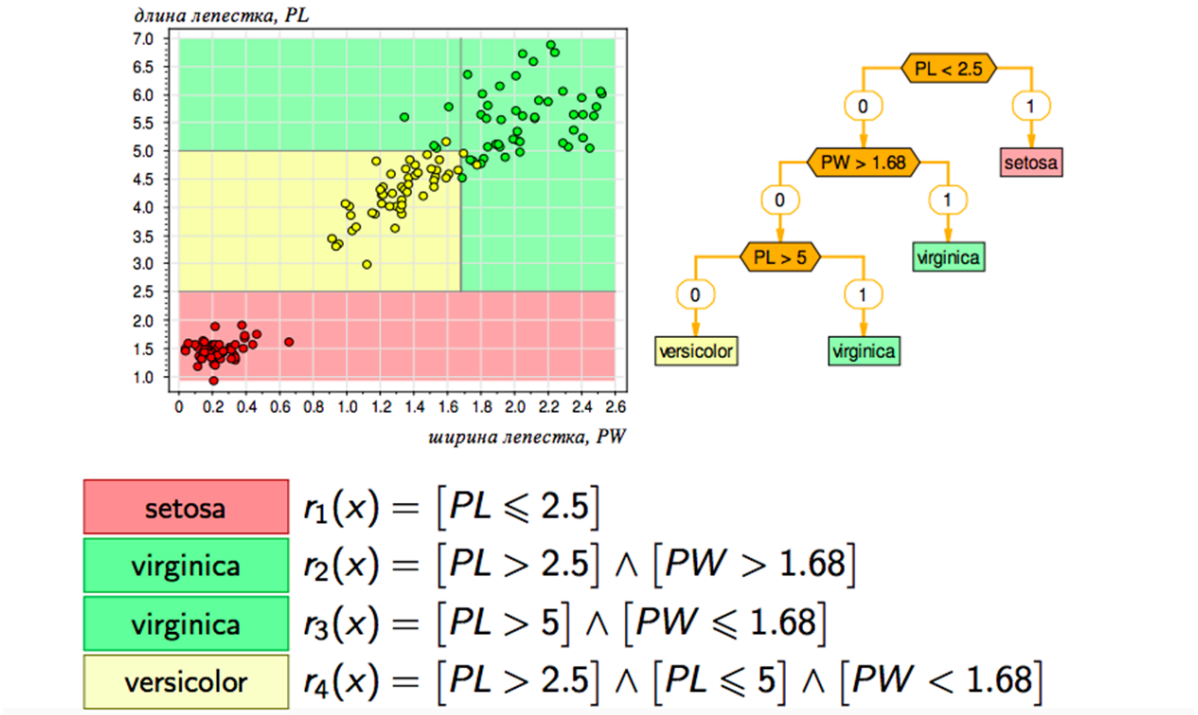
	Может параллелизоваться лишь на уровне одного дерева

6. Реализация градиентного бустинга в Python

Рассмотрим различные техники ансамблирования в машинном обучении.

Повторение

Повторим теорию решающих деревьев. Дерево — некоторый граф, который содержит листья и вершины-правила, по которым мы разделяем данные.



Как понять, что разбиение хорошее?

Существуют критерии разбиения:

- Критерий Джини:

$$H(R) = \sum_{k=1}^{K} p_k(1 - p_k)$$

- Энтропия:

$$H(p) = -\sum_{k=1}^{K} p_k log_2 p_k$$

Если бы вершина m была листовой и относила все объекты к классу k, ошибка классификации:

$H(R_m) = \frac{1}{N_m} \sum_{x_t \in R_m} [y_i \neq k_m]$ показывает, сколько объектов предсказано неверно.

Критерий информативности при ветвлении вершины m: (l и r — правые и левые вершины)

$$Q(R_m, j, s) = H(R_m) - \frac{N_l}{N_m} H(R_l) - \frac{N_r}{N_m} H(R_r) \to max_{j,s}$$

Преимущества решающих деревьев:

- хорошо интерпретируются;
- легко обобщаются для регрессии и классификации;
- допускаются разнотипные данные.

Недостатки:

- сравнение с линейными алгоритмами на линейно разделимой выборке — фиаско;
- переобучение;
- неустойчивость к шуму, составу выборки, критерию.

Методы ансамблирования

Идея ансамблирования:

- Объединить несколько моделей, получить от них ответ на загруженные данные и сделать общий вывод.
- Построить из множества плохих алгоритмов один хороший.

Пусть у нас есть ансамбль базовых алгоритмов:

$$a(x) = C\left(F\left(b_1, \ldots, b_t, x\right)\right),$$

где:

- b_i — базовый алгоритм,
- F — агрегирующая функция,
- C — решающее правило.

Базовые алгоритмы, решающие одну и ту же задачу, часто бывают взяты из одной модели.

Bagging

Чтобы повысить разнообразие базовых алгоритмов, можно воспользоваться методами стохастического ансамблирования, например, Bagging — Bootstrap Aggregating.

В бэггинге выборки для каждого базового алгоритма образуются с помощью подвыборки с возвращением. В этом случае некоторые объекты могут попасть в подвыборку 2 или 3 раза, а некоторые — вовсе не попасть. На каждую такую подвыборку приходится примерно 63% от данных всего дерева. Такое разделение позволяет сделать деревья подвыборок более независимыми.

Модель случайного леса — бэггинг для деревьев.

Сравним одно дерево и модель случайного леса:

```
%matplotlib inline

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

from sklearn.tree import DecisionTreeRegressor

from sklearn.ensemble import BaggingRegressor, GradientBoostingRegressor

import matplotlib.pyplot as plt
```

```
from mlxtend.plotting import plot_decision_regions
import matplotlib.gridspec as gridspec
import itertools


from sklearn.linear_model import LogisticRegression
from sklearn.naive_bayes import GaussianNB
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import SVC
from sklearn import datasets
import numpy as np
from sklearn import tree
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score


import mlxtend
from sklearn.datasets import make_moons, make_circles
```

```
from sklearn.model_selection import train_test_split


def make_sunny_moons(n_sun=50, n_moons=100, noise=0.0, sun_radius=1.9,
theta=None):
    X_moons, y_moons = make_moons(n_samples=n_moons, noise=noise)
```

```
    if not n_sun:
        return X_moons, y_moons

    np.random.seed(0xBEEFBAD)
    angles = np.arange(0, 2 * np.pi, 2 * np.pi / n_sun)
    X_sun = sun_radius * np.column_stack([np.cos(angles), np.sin(angles)]) + np.array([0.5, 0.25])
    X_sun += np.random.normal(scale=noise, size=X_sun.shape)
    y_sun = 2 * np.ones(n_sun)

    X = np.vstack([X_moons, X_sun])
    y = np.concatenate([y_moons, y_sun]).astype(int)
    X -= X.mean(axis=0)

    if theta is None:
        theta = np.pi / 4
    c, s = np.cos(theta), np.sin(theta)
    R = np.array(((c,-s), (s, c)))
    X = X @ R

    return X, y
```

```
X, y = make_sunny_moons(n_sun=150, n_moons=300, noise=0.15)

X_train, X_test, y_train, y_test = train_test_split(X, y)
```

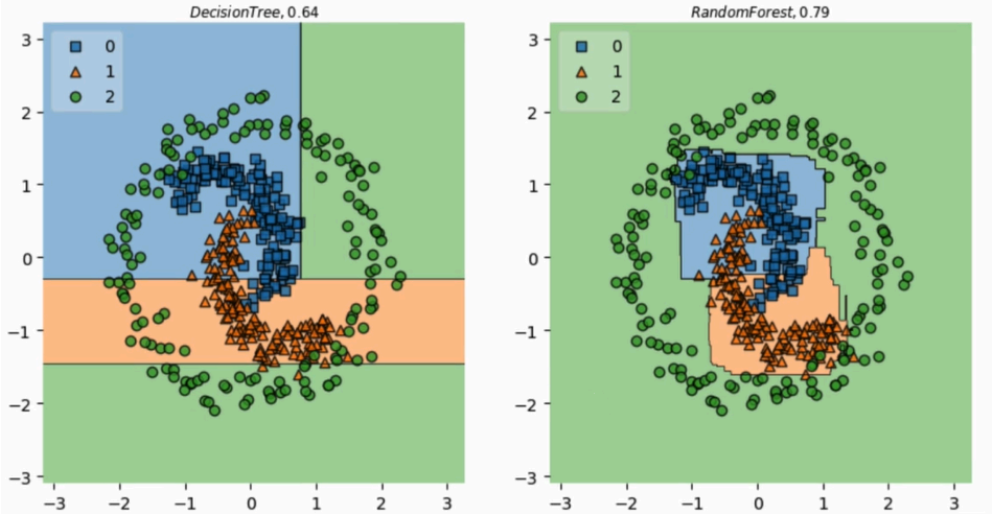
Скрипт для запуска функции make_sunny_moons, если ее нет:

```
# !pip install mlxtend==0.19.0 --upgrade --no-deps
```

```
import mlxtend

mlxtend.__version__
```

```
clf0 = tree.DecisionTreeClassifier(random_state=1, max_depth=2)

clf1 = RandomForestClassifier(random_state=1, n_estimators=100, max_depth=3)


gs = gridspec.GridSpec(1, 2)


fig = plt.figure(figsize=(10,5))


labels = ['Decision Tree ' ,'Random Forest ']

for clf, lab, grd in zip([clf0,  clf1,],
                         labels,
                         itertools.product([0, 1], repeat=2)):
```

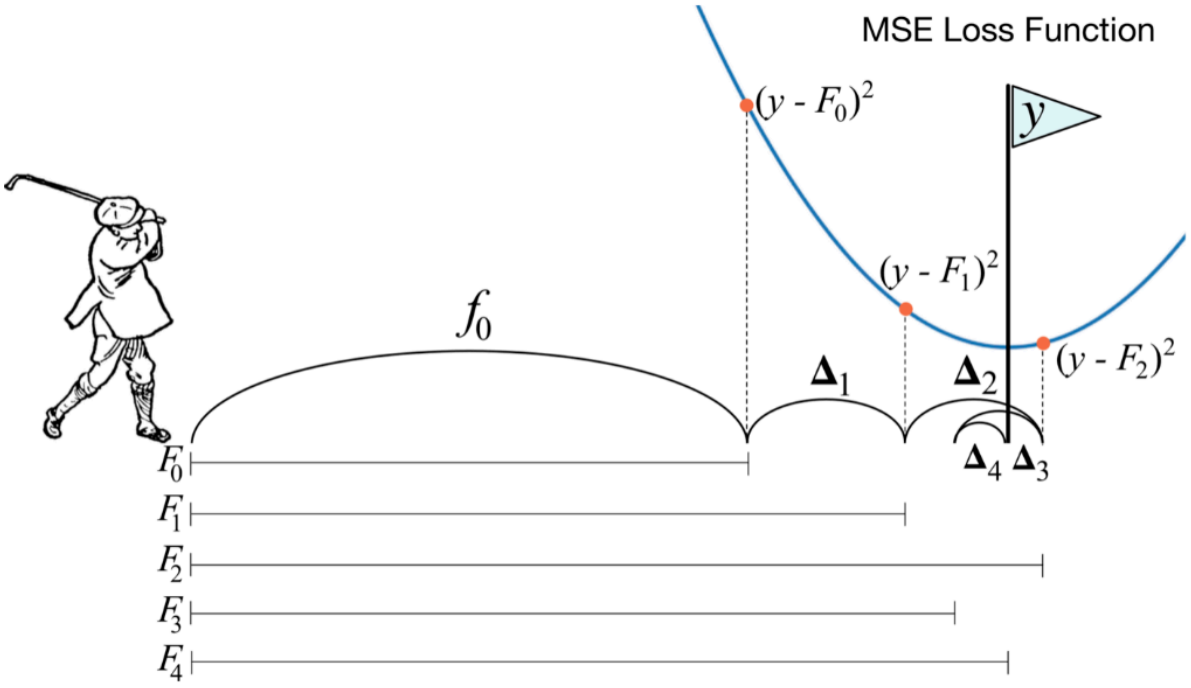
```
    clf.fit(X_train, y_train)
    ax = plt.subplot(gs[grd[0], grd[1]])
    a = accuracy_score(y_test, clf.predict(X_test))

    fig = plot_decision_regions(X=X, y=y, clf=clf, legend=2)
    plt.title('$_{'+str(lab)+ ', ' + str('{:.2f}'.format(a))+'}$')

plt.show()
```



Видим, что модель случайного леса отрабатывает классификацию лучше.

Boosting

Все решающие деревья в бэггинге имели один и тот же вес, влияли на ответ одинаково. Есть другой способ ансамблирования — boosting. Идея бустинга — обучать базовые алгоритмы на тех примерах, с которыми не справился предыдущий алгоритм.



Рассмотрим алгоритм бустинга на примере задачи регрессии.

Нам нужно найти такой алгоритм $a_N(x) = \sum b_i(x)$, который бы минимизировал функционал ошибки для задачи регрессии. В нашем случае мы минимизируем квадратичную ошибку:

$$\frac{1}{2}\sum_i \left(a\left(x_i\right) - y_i\right)^2 \to min$$

Первый алгоритм:

$$b_1(x) = argmin_{b \in A} \frac{1}{2}\sum_i \left(b\left(x_i\right) - y_i\right)^2$$

Строим функцию сдвига:

$$s_i = y_i - b_1\left(x_i\right)$$

Второй алгоритм:

$$b_2(x) = argmin_{b \in A} \frac{1}{2} \sum_i \left(b\left(x_i\right) - s_i\right)^2$$

$$b_1 - s_i = b_1 - \left(y_i - b_2\right) = b_1 + b_2 - y_i$$

Градиентный бустинг

Теперь запишем общий случай градиентного бустинга.

Есть ансамбль алгоритмов:

$$a_N(x) = \sum \gamma_i b_i(x)$$

Минимизируем функцию потерь:

$$\sum_{i=1} L\left(y_i, a_N\left(x_i\right)\right) \to min_{b_N, \gamma_N}$$

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

$$\sum_{i=1} L\left(y_i, a_{N-1}\left(x_i\right) + s_i\right) \to min_{s_1, \ldots, s_l}$$

Сдвиги находятся следующим образом:

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

$$b_N(x) = argmin_{b \in A} \sum_i \left(b\left(x_i\right) - s_i\right)^2$$

$$\gamma_N = argmin_\gamma \sum_{i=1} L\left(y_i, a_{N-1}\left(x_i\right) + \gamma b_N\right)$$

Adaboost (Adaptive boosting)

Для Adaboost реализуется следующий алгоритм:

1. Задают значения весов $\omega_i = \frac{1}{N}$, $i = 1, 2, \ldots, N$.

2. Для $m = 1, \ldots, M$:
   a. Находим классификатор $G_m(x)$ для веса ω_i.
   b. В Adaboost возникает экспоненциальная функция потерь $err_m = \frac{\sum_{i=1}^{N} \omega_i I(y_i \neq G_m(x_i))}{\sum_{i=1}^{N} \omega_i}$.
   c. $\alpha_m = log\left(\frac{1-err_m}{err_m}\right)$.
   d. $\omega_i \leftarrow \omega_i \cdot exp[\alpha_m \cdot I(y_i \neq G_m(x_i))]$, $i = 1, 2, \ldots, N$.
3. $G(x) = sign\left[\sum_{m=1}^{M} \alpha_m G_m(x)\right]$.

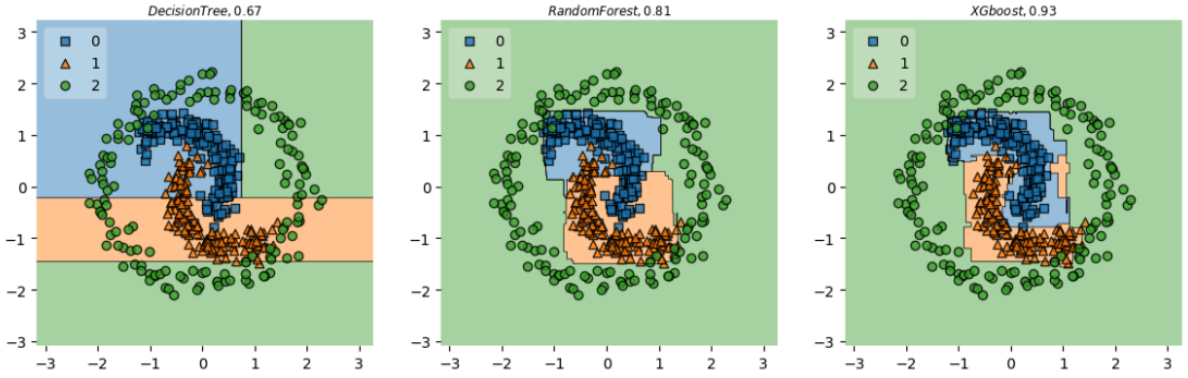
Рассмотрим бустинг тоже на деревьях:

```
import xgboost as xgb

from xgboost import XGBClassifier


model = XGBClassifier()

model.fit(X_train, y_train)


y_pred = model.predict(X_test)

predictions = [round(value) for value in y_pred]

accuracy = accuracy_score(y_test, predictions)

print("Accuracy: %.2f%%" % (accuracy * 100.0))
```

>>> Accuracy: 94.69%

Сравним xgboost с алгоритмами, которые использовали ранее:

```
clf0 = tree.DecisionTreeClassifier(random_state=1, max_depth=2)

clf1 = RandomForestClassifier(random_state=1, n_estimators=100, max_depth=3)

clf2 = XGBClassifier()


gs = gridspec.GridSpec(1, 3)


fig = plt.figure(figsize=(14,4))


labels = ['Decision Tree ' ,'Random Forest ', 'XGboost']

for clf, lab, grd in zip([clf0,  clf1, clf2],

                         labels,

                         itertools.product([0, 1,2], repeat=2)):


    clf.fit(X_train, y_train)

    ax = plt.subplot(gs[grd[0], grd[1]])

    a = accuracy_score(y_test, clf.predict(X_test))


    fig = plot_decision_regions(X=X, y=y, clf=clf, legend=2)

    plt.title('$_{'+str(lab)+ ', ' + str('{:.2f}'.format(a))+'}$')


plt.show()
```



Видим, что xgboost отработал лучше других алгоритмов.

Библиотека CatBoost

CatBoost — удобная библиотека для бустинга, в которой не надо заботиться, какого типа имеющиеся данные.

Рассмотрим пример:

```
#!pip install catboost
```

```
from catboost import CatBoostClassifier, Pool


train_data = Pool(
  [
    [[0.1, 0.12, 0.33], [1.0, 0.7], 2, "male"],
    [[0.0, 0.8, 0.2], [1.1, 0.2], 1, "female"],
    [[0.2, 0.31, 0.1], [0.3, 0.11], 2, "female"],
    [[0.01, 0.2, 0.9], [0.62, 0.12], 1, "male"] #обучающие данные для catboost
```

```
    ],
    label = [1, 0, 0, 1],
    cat_features=[3],
    embedding_features=[0, 1]
)

eval_data = Pool(
    [
        [[0.2, 0.1, 0.3], [1.2, 0.3], 1, "female"],
        [[0.33, 0.22, 0.4], [0.98, 0.5], 2, "female"],
        [[0.78, 0.29, 0.67], [0.76, 0.34], 2, "male"],
    ], #тестирование
    label = [0, 1, 1],
    cat_features=[3],
    embedding_features=[0, 1]
)

model = CatBoostClassifier(iterations=10)

model.fit(train_data, eval_set=eval_data)
preds_class = model.predict(eval_data)
```

```
>>> bestTest = 0.6772319605
```

```
bestIteration = 9
```

```
preds_class
```

```
>>> array([0, 0, 1])
```

Дополнительные материалы для самостоятельного изучения

1. Alex Rogozhnikov, Gradient Boosting explained
2. CatBoost, XGBoost и выразительная способность решающих деревьев