



Лекция 13: Бустинг

«Самый древний» дискретный бустинг

■ Усиление за счет фильтрации (Schapire 1989):

- Случайно выбрать $n_1 < n$ примеров из выборки D без удаления
- Получить D_1 и построить «слабую» модель $b_1(x)$
- Выбрать $n_2 < n$ примеров из D с фильтрацией по $b_1(x)$ - применить к примерам $b_1(x)$, и с вероятностью $1/2$ добавлять ошибки в D_2 и с вероятностью $1/2$ не ошибки
- На D_2 , где половина – ошибки $b_1(x)$, построить «слабую» модель $b_2(x)$
- Выбрать все D_3 из D где $b_1(x)$ и $b_2(x)$ дают разный прогноз и построить «слабую» модель $b_3(x)$
- Финальный «усиленный» классификатор – голосование $b_1(x), b_2(x), b_3(x)$

■ Почему работает?

- в голосовании должно быть 2+ голосов
- $b_1(x)$ и $b_2(x)$ - максимально не похожи
- если $b_1(x) \neq b_2(x)$, то решает $b_3(x)$, который для этого учился
- если $b_1(x) = b_2(x)$, то мнение $b_3(x)$ не интересно

Идея адаптивного бустинга

- Развивает идею «старого» бустинга с фильтрацией:
 - Вместо случайной выборки – перевзвешивание, т.е. оценка **«важности»** или **«сложности»** **примеров** (зависит от ошибки ансамбля на примере)
 - Построение «слабых» классификаторов (точность хотя бы $>50\%$)
 - Результирующий «сильный» классификатор может иметь много слабых классификаторов и пользуется взвешенным голосованием, здесь **вес базового классификатора – его «сила»**
- Заимствует постановку задачи и подход у Forward Stagewise Additive Modeling (FSAM), решающего задачу прогнозирования:
 - Выборка $Z = \{(x_i, y_i)\}_{i=1}^l \in X \times Y$
 - Модель – взвешенный с весами α_i ансамбль M базовых моделей $a(x) = \sum_m \alpha_m b_m(x)$, который строится последовательно $a_m(x) = a_{m-1}(x) + \alpha_m b_m(x)$, где ищутся α_m, b_m при фиксированном a_{m-1}
 - Функция потерь $L: Y \times Y \rightarrow \mathbb{R}^+$

Forward Stagewise Additive Modeling

■ FSAM

- строит модель **последовательно**, добавляя к текущей модели a_m на шаге m «лучший» b_m с «лучшим» весом α_m :
- Инициализация $a_0(x) = 0$
- Цикл M итераций по m :

$$(b_m, \alpha_m) = \underset{b, \alpha}{\operatorname{argmin}} \sum_{i=1}^l L(y_i, a_{m-1}(x_i) + \alpha b(x_i))$$
$$a_m(x) = a_{m-1}(x) + \alpha_m b_m(x)$$

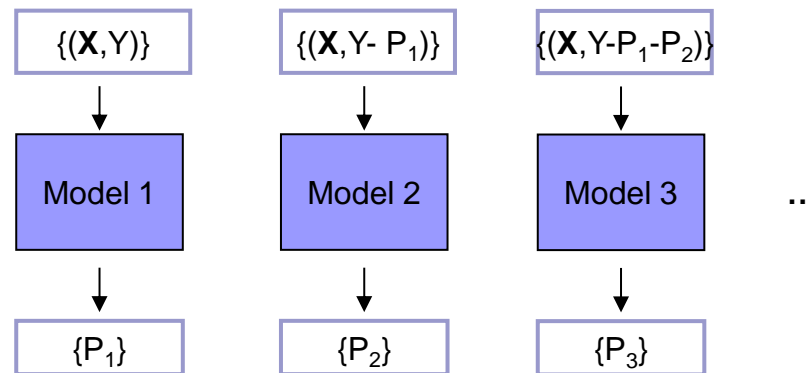
- Для некоторых L и b_i - простое аналитическое решение, примеры:

- **AdaBoost**: L – экспоненциальная функция потерь, b_i - простое дерево (или даже «пень»), α_i вычисляются аналитически
- **Additive Groves**: L – квадратичная, а b_i - дерево, $\alpha_i = 1$, получаем ансамбль на остатках, для борьбы с переобучением используется bagging и специальный алгоритм, ищущий «оптимальное» сочетание сложности ансамбля и индивидуальных моделей

Additive (Tree) Groves

- Особенности метода:

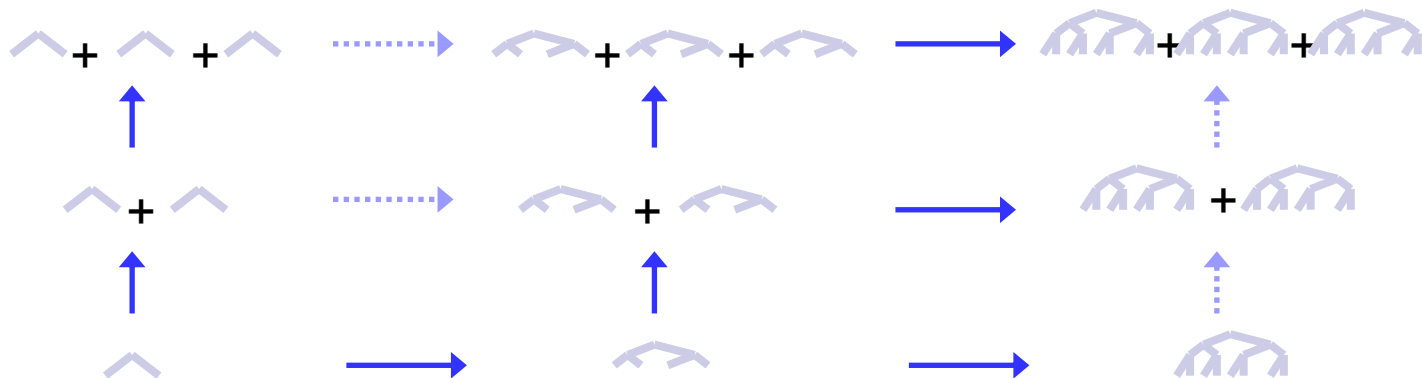
- Аддитивный ансамбль (без весов) $a(x) = \sum_m b_m(x)$, деревьев решений $b_m(x)$, построенных на $Z_m = \{(x_i, y_i - a_{m-1}(x_i))\}_i^l$, где вместо отклика используются остатки от предыдущего ансамбля.



- Поскольку обучение на остатках, то на сложных b_m быстро переобучается при увеличении, исчерпываются остатки $\{(x_i, 0)\}_i^{n < l}$
- Использует бутстрепинг подвыборку меньшего размера Z_m^* (как в random forest)
- специальный алгоритм для контроля сложности на основе роста дерева «по слоям»

Контроль сложности Additive Groves

- Переборный алгоритм контроля сложности:
 - От простого (одно маленькое дерево) делаются шаги в сторону увеличения ансамбля без изменения сложности деревьев (увеличивается параметр M – размер ансамбля) и в сторону увеличения глубины деревьев без увеличения ансамбля (меняется параметр обрубания или число листьев D)
 - Если качество по OOB улучшается, то шаг принимается
 - Осуществляется поиск «оптимальной» точки гиперпараметров (N, D) , без повторных вычислений (если пришли в точку разными путями)



Адаптивный дискретный бустинг

■ Основные допущения:

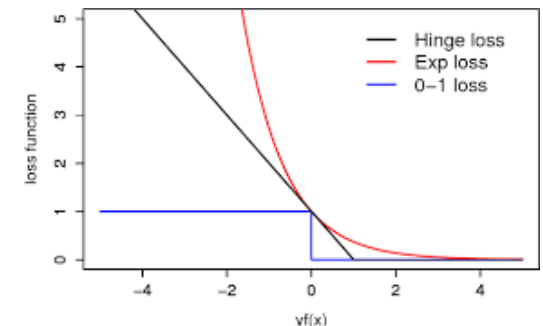
- Выборка $Z = \{(x_i, y_i)\}_{i=1}^l \in X \times \{-1, +1\}$, где у каждого наблюдения (x_i, y_i) свой вес $w_i \geq 0$, причем $\sum_i w_i = 1$ (распределение)
- Модель-ансамбль с взв. голосованием $a(x) = \text{sign}[\sum_m \alpha_m b_m(x)]$
- Введем понятие взвешенной (по распределению w_i) ошибки классификации:

$$\text{Err}_w(a(x)) = \sum_{i=1}^l w_i I[y_i \neq a(x_i)]$$

- Будем использовать экспоненциальную функцию потерь $L_{\text{exp}}(y, a(x)) = \exp(-y \sum_m \alpha_m b_m(x))$

■ Идея алгоритма обучения (цикл):

1. считаем ошибки для всех примеров
2. перевзвешиваем все примеры
3. обучаем базовый классификатор
4. добавляем его в ансамбль с новым весом



AdaBoost (алгоритм)

- Инициализация:

- $a_0(x) = 0, \forall i: w_i^{(1)} = 1/l$

- Итераций по m от 1 до M :

- Строим **слабый классификатор** $b_m(x)$, минимизирующий и допускающий ошибку $Err_{w^{(m)}}(b_m) < 0.5$

$$b_m(x) = \operatorname{argmin}_b \sum_{i=1}^l w_i^{(m)} I[b(x_i) \neq y_i]$$

- Добавляем b_m в ансамбль $a_m(x) = a_{m-1}(x) + \alpha_m b_m(x)$, где **силу классификатора** вычисляем как:

$$\alpha_m = \frac{1}{2} \log \left(\frac{(1 - Err_{w^{(m)}}(b_m))}{Err_{w^{(m)}}(b_m)} \right)$$

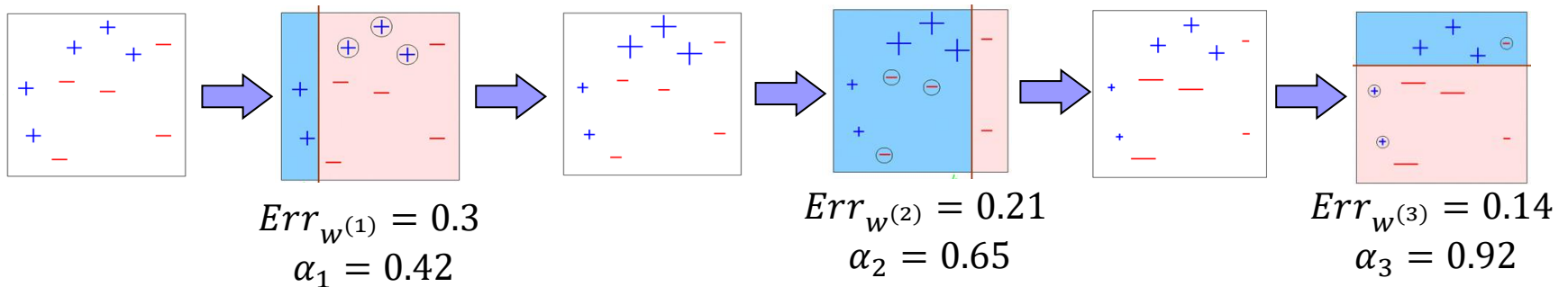
- Пересчитываем и нормируем **веса важности** примеров:

$$\tilde{w}_i = w_i^{(m)} \exp(-y_i \alpha_m b_m(x_i)), w_i^{(m+1)} = \tilde{w}_i / \sum_j \tilde{w}_j$$

- Откуда взялись формулы для b_m, α_m и $w^{(m)}$?

Adaboost (демо)

- Простой пример Adaboost (длины 3) на «пнях» (деревьях глубины 1):



- Финальный классификатор:

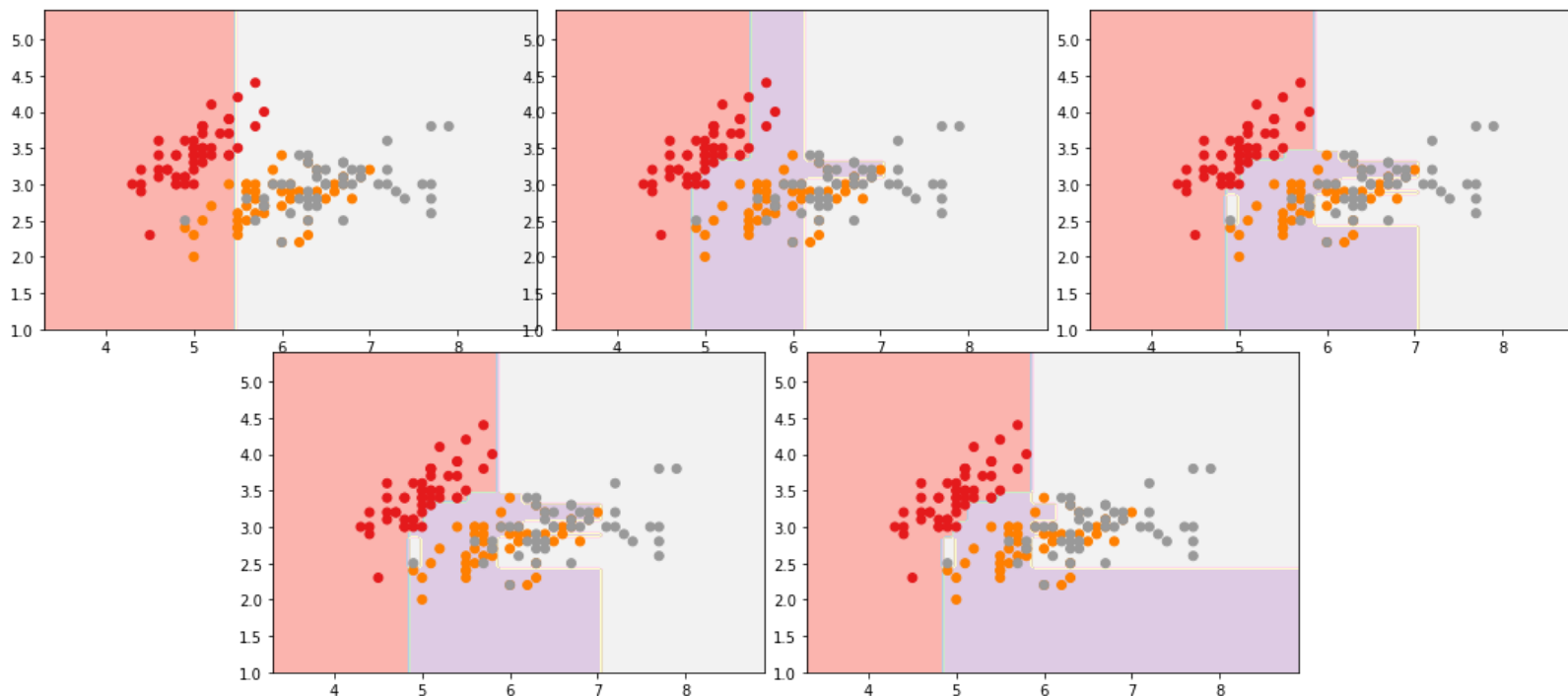
$$a(x) = \text{sign} \left(0.42 * \begin{array}{|c|} \hline \text{blue} \\ \hline \end{array} + 0.65 * \begin{array}{|c|} \hline \text{blue} \\ \hline \end{array} + 0.92 * \begin{array}{|c|} \hline \text{blue} \\ \hline \end{array} \right)$$

Пример

```
from sklearn.ensemble import AdaBoostClassifier
from sklearn.tree import DecisionTreeClassifier

weak_learner = DecisionTreeClassifier(max_leaf_nodes=2)

for i in range(1,50,10):
    adaboost_clf = AdaBoostClassifier(estimator=weak_learner, n_estimators=i,
                                      algorithm="SAMME", random_state=42).fit(X, y)
    DecisionBoundaryDisplay.from_estimator(adaboost_clf, X, cmap="Pastell")
    plt.scatter(*X.T, c=y, cmap="Set1")
```



Пересчет весов классификаторов

- Рассмотрим на итерации m алгоритма эмпирический риск с функцией потерь на основе ошибки классификации, он ограничен сверху риском с экспоненциальной функцией потерь:

$$Q_{perc}^{(m)} = \sum_{i=1}^l I[y_i \neq a_m(x_i)] \leq Q_{exp}^{(m)} = \sum_{i=1}^l \exp[-y_i a_m(x_i)] =$$

$$= \sum_{i=1}^l \exp \left[-y_i \left(\sum_{j=1}^m \alpha_j b_j(x_i) \right) \right] = \sum_{i=1}^l \underbrace{\exp \left[-y_i \left(\sum_{j=1}^{m-1} \alpha_j b_j(x_i) \right) \right]}_{\sim w_i^{(m)} - \text{не зависит от } \alpha_m, b_m} \exp[-y_i \alpha_m b_m(x_i)] \sim$$

$$\sim \sum_{i=1}^l w_i^{(m)} \exp[-y_i \alpha_m b_m(x_i)] = e^{-\alpha_m} \underbrace{\sum_{i: y_i = b_m(x_i)} w_i^{(m)}}_{\text{доля верных прогнозов с весами } (1 - Err_w^{(m)})} + e^{\alpha_m} \underbrace{\sum_{i: y_i \neq b_m(x_i)} w_i^{(m)}}_{\text{доля ошибок с весами } Err_w^{(m)}} =$$

$$= \dots$$

$$e^{-\alpha b} = e^{-\alpha} I[b = 1] + e^{\alpha} I[b = -1]$$

доля верных прогнозов
с весами $(1 - Err_w^{(m)})$

доля ошибок с
весами $Err_w^{(m)}$

Пересчет весов классификаторов

- Продолжение:

$$\dots = (1 - Err_{w^{(m)}})e^{-\alpha_m} + (Err_{w^{(m)}})e^{\alpha_m} = Q_{Err_{w^{(m)}}}(\alpha_m) \rightarrow \min$$

- Найдем $\min_{\alpha_m} [Q_{Err_{w^{(m)}}}(\alpha_m)]$:

$$\frac{\partial Q_{Err_{w^{(m)}}}}{\partial \alpha_m} = 0 \Rightarrow \alpha_m = \frac{1}{2} \log \left(\frac{1 - Err_{w^{(m)}}}{Err_{w^{(m)}}} \right)$$

- Подставляем α_m в $Q_{Err_{w^{(m)}}}$ и получаем, что верхняя оценка эмпирического риска экспоненциально уменьшается с уменьшением взвешенной ошибки $Err_{w^{(m)}}$:

$$\begin{aligned} (1 - Err_{w^{(m)}})e^{-\frac{1}{2} \log \left(\frac{1 - Err_{w^{(m)}}}{Err_{w^{(m)}}} \right)} + (Err_{w^{(m)}})e^{\frac{1}{2} \log \left(\frac{1 - Err_{w^{(m)}}}{Err_{w^{(m)}}} \right)} = \\ = 2\sqrt{Err_{w^{(m)}}(1 - Err_{w^{(m)}})} \leq \exp \left(-2(0.5 - Err_{w^{(m)}})^2 \right) \end{aligned}$$

Обучение слабых классификаторов и пересчет весов

- При фиксированном $\alpha_m > 0$ выразим наилучший слабый b_m :

$$\begin{aligned} & e^{-\alpha_m} \sum_{i: y_i = b(x_i)} w_i^{(m)} + e^{\alpha_m} \sum_{i: y_i \neq b(x_i)} w_i^{(m)} = \\ & = [(e^{\alpha_m} - e^{-\alpha_m}) \sum_i w_i^{(m)} I[y_i \neq b(x_i)] + e^{-\alpha_m} \sum_i w_i^{(m)}] \rightarrow \min_{b_m} \end{aligned}$$

$$b_m(x) = \operatorname{argmin}_b \sum_{i=1}^l w_i^{(m)} I[b(x_i) \neq y_i]$$

Не зависит от b_m и > 0 при $\alpha_m > 0$

- Поскольку $a_m(x) = a_{m-1}(x) + \alpha_m b_m(x)$ потери выразим рекурсивно:

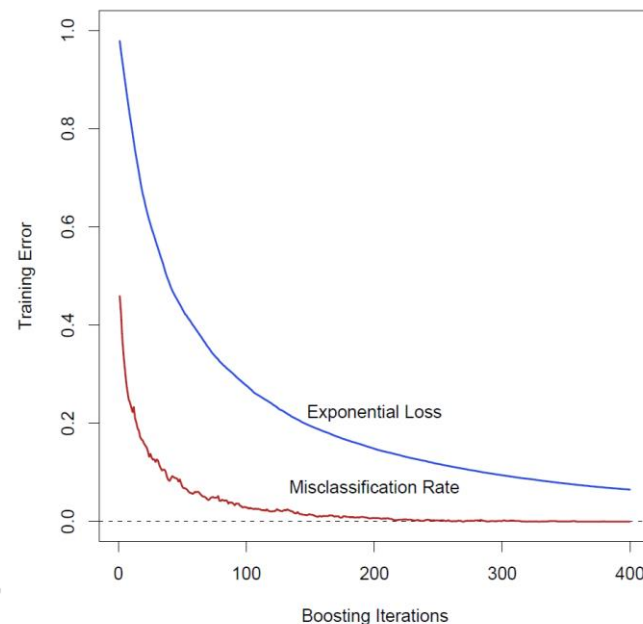
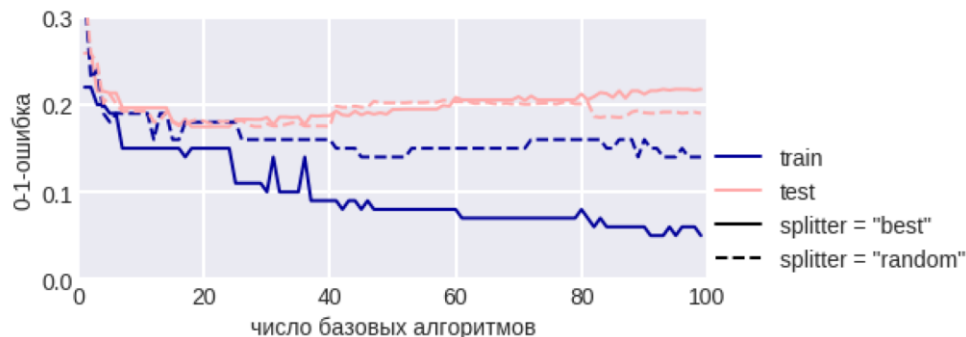
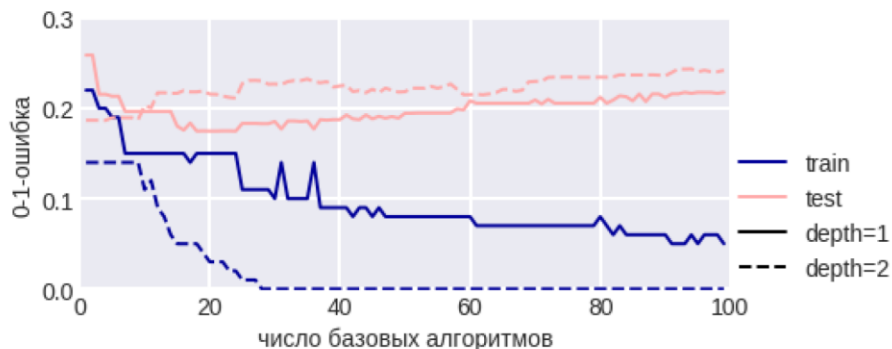
$$\begin{aligned} & \sum_{i=1}^l \exp[-y_i a_m(x_i)] = \sum_{i=1}^l \exp \left[-y_i \left(\sum_{j=1}^m \alpha_j b_j(x_i) \right) \right] = \\ & = \sum_{i=1}^l w_i^{(m)} \exp[-y_i \alpha_m b_m(x_i)] = \sum_{i=1}^l w_i^{(m-1)} \exp[-y_i (\alpha_{m-1} b_{m-1}(x_i) + \alpha_m b_m(x_i))] = \\ & = \sum_{i=1}^l w_i^{(1)} \exp[-y_i \alpha_1 b_1(x_i)] \exp[-y_i \alpha_2 b_2(x_i)] \dots \exp[-y_i \alpha_m b_m(x_i)] \Rightarrow \\ & \Rightarrow w_i^{(m+1)} = w_i^{(m)} \exp(-y_i \alpha_m b_m(x_i)) \end{aligned}$$

Теоретические результаты

- Условия применимости:
 - «достаточно богатое» семейство слабых классификаторов
 - качество слабого классификатора лучше случайного прогноза
 - «не слишком высокая» зашумленность данных
- Формально доказан ряд важных свойств алгоритма Adaboost:
 - приводит к минимизации эмпирического риска с экспоненциальной функцией потерь при использовании предложенной процедуры пошагового усложнения ансамбля и полученных формул пересчета нормированных весов примеров и весов слабых классификаторов
 - сходится за конечное число шагов
 - с увеличением числа итераций увеличивает зазор между классами и, значит, уменьшает ошибку классификации – полезно строить график зазора от шага итерации и распределение отступов на каждом шаге

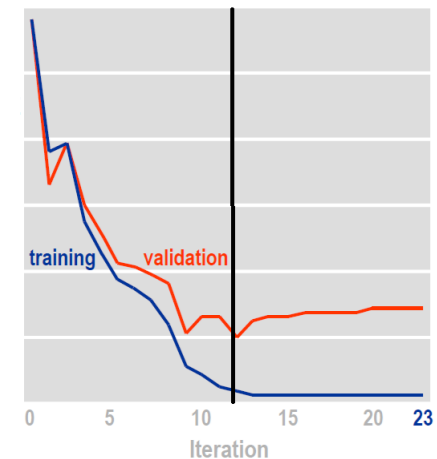
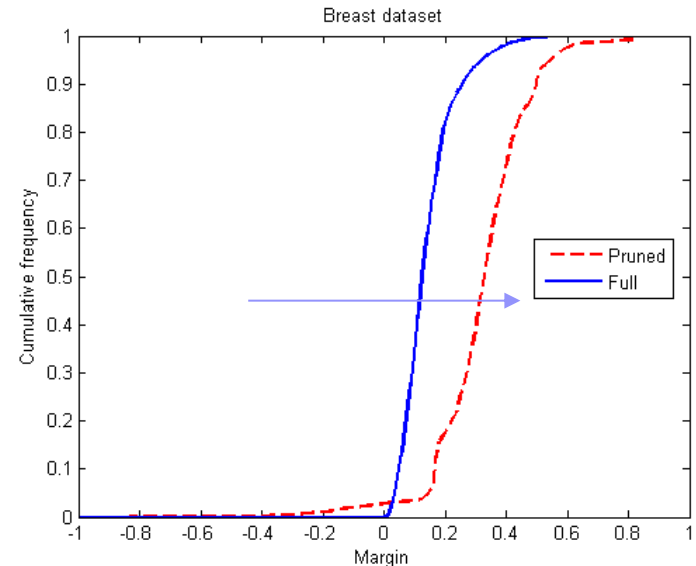
Сложность ансамбля Adaboost и борьба с переобучением

- Сложность ансамбля:
 - Определяется размером ансамбля (чем больше тем сложнее)
 - Определяется сложностью базового классификатора
 - За счет экспоненциальной функции потерь может улучшаться на проверочной выборке даже когда ошибка на тренировочной ушла в 0



Борьба с переобучением

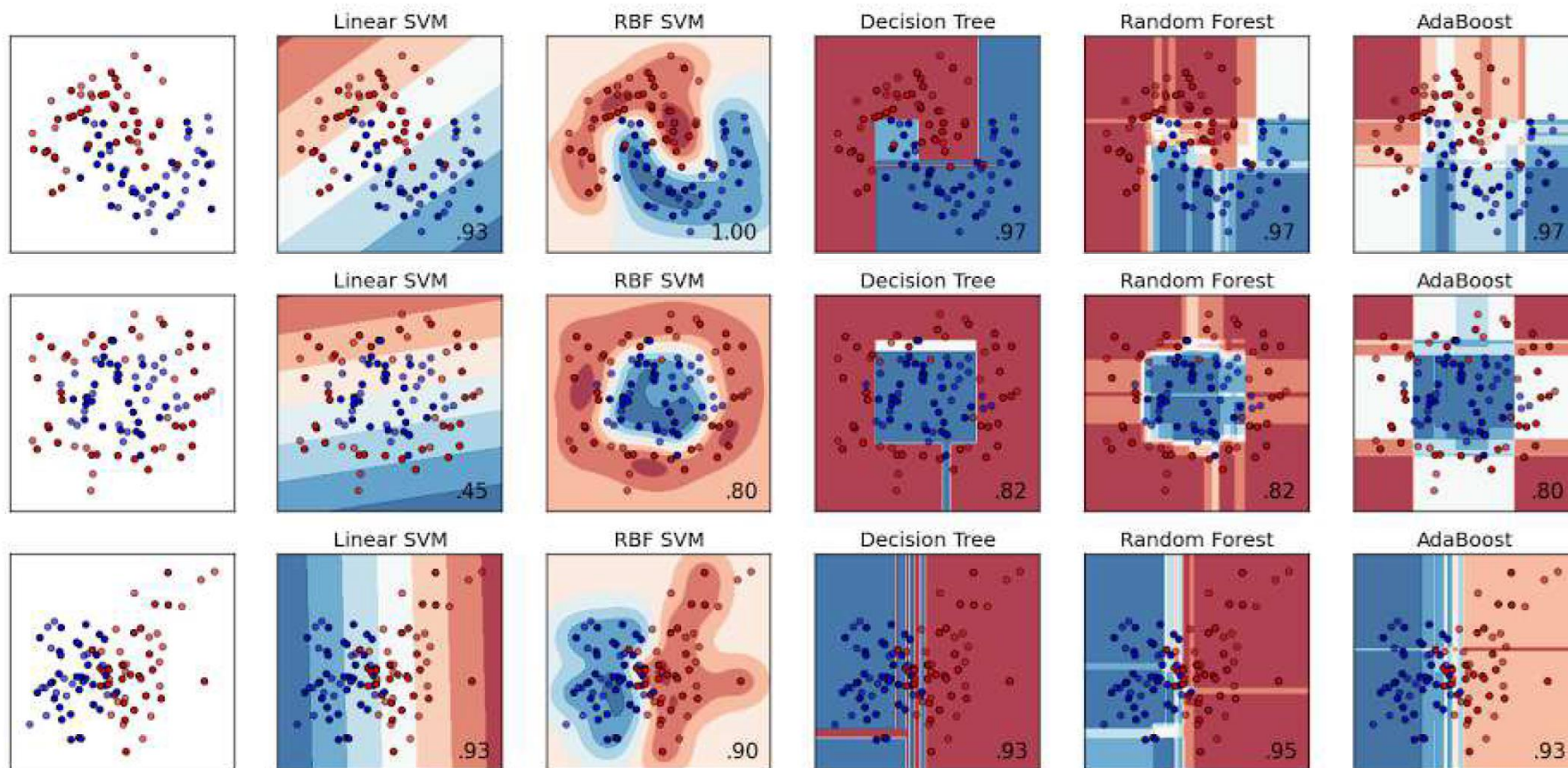
- Методы борьбы с переобучением:
 - Управлять размером ансамбля
 - Ограничивать сложность или упрощать (например pruning) слабые классификаторы
 - Можно пытаться контролировать (менять вес) или удалять выбросы «на лету», анализируя веса или отступы и их распределение (оно с каждой итерацией должно «смещаться вправо», «раздвигая» классы)
 - В некоторых реализациях можно контролировать learning rate или использовать регуляризацию
 - Т.к. бустинг ищет зависимости от простых к сложным, то можно использовать early stopping



Недостатки Adaboost

- Чувствительность к выбросам:
 - из-за экспоненциальной функции потерь может быстро переобучаться при наличии шума
 - можно пытаться контролировать выбросы «на лету»
 - нужно использовать простые слабые модели, в результате не позволяет строить маленькие ансамбли сложных моделей, только большие ансамбли простых моделей
- Проблемы применения:
 - Вычислительно сложная модель (если много базовых)
 - Не интерпретируемая модель, даже если базовые модели интерпретируются
- Проблемы обучения:
 - Склонен к переобучению в зашумленных задачах
 - Требуются большие выборки
 - Плохо распараллеливается (только «внутри» обучения базовой модели)

Сравнение Adaboost



Бустинг с перевыборкой (пример)

- Arc-x4 (Breiman, 1996) как в AdaBoost последовательно строит модель $a_m(x) = a_{m-1}(x) + \alpha_m b_m(x)$, но:
 - В классической версии **нет весов классификаторов** – голосующий комитет или усреднение (хотя есть версии с эвристическим или FSAM пересчетом весов)
 - слабый классификатор **не учитывает веса примеров**, а строится на случайной выборке, где у наблюдения вероятность попасть в нее пропорциональна весу
 - Не требуется взвешенная функция ошибки или функция потерь – **может «бустить» любые базовые модели**
 - Вес имеет семантику как и в AdaBoost – **«сложность» примера**, и пересчитывается в зависимости от числа ошибок на нем слабых классификаторов ансамбля, классическая формула для классификации (эвристика):

$$w_i^{(m)} = \frac{(1 + \sum_{s=1}^m I[b_s(x_i) \neq y_i])^4}{\sum_{j=1}^l w_j^{(m)}}$$

Алгоритм Arc-X4

- Инициализация: $a_0(x) = 0, \forall i: w_i^{(1)} = 1/l$
- Цикл алгоритма (M итераций):
 - Формируется из набора Z **случайная выборка** $Z^{(m)}$ размера n , с вероятностью выбора примера пропорциональной распределению весов примеров ($n < l$ – параметр): $P(x_i \in Z^{(m)}) \propto w_i^{(m)}$
 - Обучается **слабая модель** $b_m(x)$ со своей функцией потерь
 - Добавляется b_m в ансамбль $a_m(x)$, где **силу классификатора** не меняют (хотя есть версии с весами и регуляризацией):

для классификации: $a(x) = \operatorname{argmax}_i [b_i(x)]$

для регрессии: $a(x) = \frac{1}{M} \sum b_i(x)$

- Пересчитываются и нормируются **веса важности** примеров в зависимости от числа ошибшихся слабых классификаторов:

$$w_i^{(m)} = \frac{(1 + \sum_{s=1}^m L[b_s(x_i), y_i])^4}{\sum_{j=1}^l w_j^{(m)}}$$

Arc-x4

■ Достоинства:

- Простота и эффективность (сопоставим с AdaBoost по качеству)
- Не накладывает требования на тип моделей и функции потерь
- Можно использовать свои эвристики весов
- Легко адаптируется к разным задачам прогнозирования

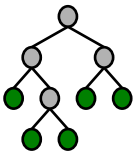
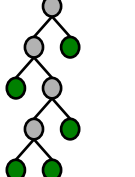
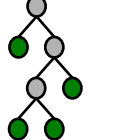
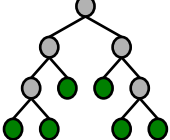
■ Основной недостаток – нет теоретического обоснования

■ Демо-пример:

$$P(x_i \in Z^{(m)}) \propto (1 + Err_m)^4$$

$$Err_m = \sum_{s=1}^m I[b_s(x_i) \neq y_i]$$

	m=1		m=2		m=3		m=4	...
<u>x</u>	<u>w</u>	<u>Err</u>	<u>w</u>	<u>Err</u>	<u>w</u>	<u>Err</u>	<u>w</u>	
1	1	1	1.5	1	.5	2	.97	
2	1	0	.75	0	.25	0	.06	
3	1	1	1.5	2	4.25	3	4.69	
4	1	0	.75	1	.5	1	.11	
5	1	0	.75	0	.25	0	.06	
6	1	0	.75	0	.25	1	.11	

Идея градиентного бустинга

- Снова рассмотрим FSAM:

- На каждом шаге цикл последовательного формирования ансамбля $a_m(x) = a_{m-1}(x) + \alpha_m b_m(x)$ требует решения:

$$(b_m, \alpha_m) = \underset{b, \alpha}{\operatorname{argmin}} Q_m(b, \alpha), Q_m(b, \alpha) = \sum_{i=1}^l L(y_i, a_{m-1}(x_i) + \alpha b(x_i))$$

- Но не для всех L есть аналитическое решение, что делать?

- Использовать приближенное решение:

- Рассмотрим $Q_m(F_m)$ на шаге m как функцию от вектора прогнозов для всех примеров

$$F_m = [a_m(x_i)]_{i=1}^l = \left[\left(\sum_{j=1}^{m-1} \alpha_j b_j(x_i) + \alpha_m b_m(x_i) \right) \right]_{i=1}^l = [F_{m-1} + \alpha_m b_m(x_i)]_{i=1}^l$$

- F_0, F_1, \dots, F_m - последовательность приближений откликов

- Нужно минимизировать по F_m $Q_m = \sum_i L(y_i, F_m(x_i))$

- **Оптимизация не в пространстве параметров модели, а в пространстве прогнозов модели (размера всей выборки)**

Градиентный бустинг

■ Почему «градиентный»?

- Раскладываем риск Q_m в ряд Тейлора 1 порядка по вектору F_{m-1} :

$$\sum_i L(y_i, F_{m-1}(x_i) + a_m b(x_i)) \approx \sum_i L(y_i, F_{m-1}(x_i)) + a_m b(x_i) \frac{\partial L(y_i, F_{m-1})}{\partial F_{m-1}}(x_i) + \dots$$

- Итерационно (**градиентным методом с шагом α_m**) минимизируем Q по

$$F_m = F_{m-1} - \alpha_m \nabla Q_m, \text{ где } \nabla Q_m = \left[\frac{\partial Q}{\partial F_{m-1}}(x_i) \right]_{i=1}^l = \left[\frac{\partial L(y_i, F_{m-1})}{\partial F_{m-1}}(x_i) \right]_{i=1}^l$$

- В тоже время $F_m = F_{m-1} + \alpha_m b_m \Rightarrow$ нужно находить такое b_m , чтобы он **прогнозировал антиградиент функции потерь**
- Значит на каждом шаге строим слабую модель (со своей функцией потерь) $b_m(x)$ на обучающем наборе $Z_m = \{(x_i, -\nabla Q_m(x_i))\}_{i=1}^l$, с **$-\nabla Q_m$ в качестве отклика**
- Находим размер шага с помощью любого метода оптимизации для одномерной задачи, например, с помощью линейного поиска (вектор прогнозов $b_m(x_i)$ зафиксирован):

$$\alpha_m = \operatorname{argmin}_{\alpha \in R} \sum_{i=1}^N L(y_i, F_{m-1}(x_i) + \alpha b_m(x_i))$$

Градиентный бустинг деревьев

- Финальная модель (M – число итераций):

$$F_m(x) = F_{m-1}(x) + \nu \alpha_m T_m(x) = F_0 + \nu \alpha_1 T_1(x) + \nu \alpha_2 T_2(x) + \dots + \nu \alpha_M T_M(x)$$

- Каждая базовая модель – дерево:

$$T_m(x) = \sum_{R \in R_m} \gamma_R I[x \in R]$$

- где R_m - множество регионов, соответствующих листьям дерева, а γ_R - прогноз отклика в листе

- Каждая следующая модель $T_m(x)$

обучается на «псевдоостатках»

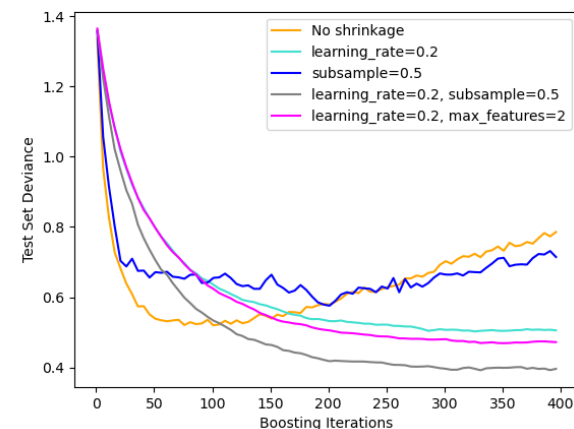
$$\text{от } F_{m-1}: y_i^{(m)} = -\frac{\partial L(y_i, F_{m-1})}{\partial F_{m-1}}(x_i)$$

- $0 < \nu < 1$ – shrinkage регуляризация



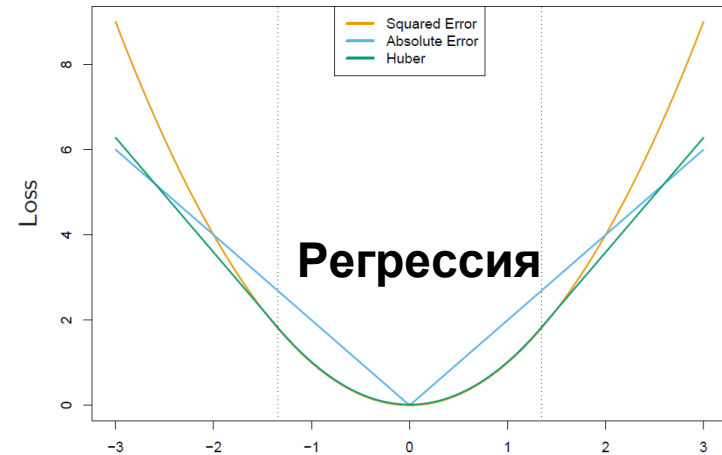
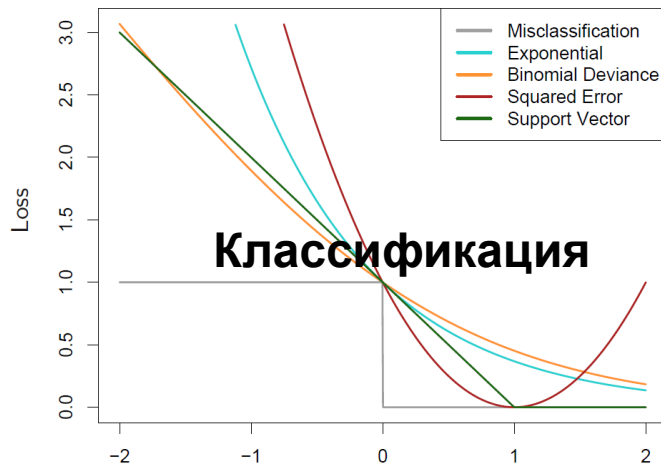
Борьба с переобучением

- Градиентный бустинг склонен к переобучению в условиях шума
- Методы борьбы с переобучением:
 - **Стохастический градиентный бустинг** - случайные подвыборки (можно использовать ООВ оценки) меньшего размера для обучения базовых моделей на каждой итерации (иногда используют бутстреппинг): $Z_m = (x_i, -\nabla Q_m(x_i))_{i=1}^{n \leq l}$
 - **Ранняя остановка** (по порогам или по контрольной выборке)
 - Контроль **размера ансамбля** и **ограничение** (или упрощение, например, pruning) **сложности базовых моделей**, также можно использовать дообучение деревьев по уровням
 - **Shrinkage** (или learning rate) – не позволяет быстро «исчерпать» псевдоостатки, но требуется больше моделей в финальном ансамбле
 - **Регуляризация** L_0, L_1 или L_2



Функции потерь

■ Кастомизируемые и робастные функции потерь:



Задача	Потери	Псевдоостаток (производная функции потерь)
Регрессия	$(y_i - a(x_i))^2$	$y_i - a(x_i)$
Регрессия	$ y_i - a(x_i) $	$\text{sign}(y_i - a(x_i))$
Регрессия	Huber	$y_i - a(x_i)$, при $ y_i - a(x_i) \leq \sigma$, иначе $\text{sign}(y_i - a(x_i))$
К классов	К классов	$I[y_i = C_k] - p_k(x_i)$ для класса k
2 класса	$\log(1 + e^{-a(x_i)y_i})$	$-y_i / (1 + e^{-a(x_i)y_i})$
...

Би- и Мульти- номинальная функция потерь

■ Классификации с несколькими классами:

- $Y = \{C_1, \dots, C_K\}$, $p_k(x) = P(Y = C_k|x)$
- Правило Байеса $k = \operatorname{argmax}_s [p_s(x)]$
- $g_k(x)$ - дискриминантная функция класса k
- $p_k(x) = \operatorname{softmax}(g_1(x), \dots, g_K(x)) = \frac{e^{g_k(x)}}{\sum_{s=1}^K e^{g_s(x)}}$
- Кросс энтропия: $L(y, p(x)) = -\sum_{k=1}^K I[y = k] \log(p_k(x)) = -\sum_{k=1}^K I[y = k] g_k(x) + \log(\sum_{s=1}^K e^{g_s(x)})$
- Псевдоостаток для класса k : $-L'(y, p_k(x)) = I[y_i = C_k] - p_k(x_i)$

■ Бинарный случай:

- $Y = \{0,1\}$, $p(x) = p_1(x) = P(Y = 1|x) = \frac{1}{1+e^{-g(x)}}$, $p_0(x) = 1 - p(x)$
- $L(y, p(x)) = y \log(p(x)) + (1 - y) \log(1 - p(x))$, и если $g(x) \equiv a(x) \Rightarrow$
 $L(y, a(x)) = \log(1 + e^{-a(x)y}) \Rightarrow$
Псевдоостаток $L'(y, a(x)) = -y/(1 + e^{-a(x)y})$

Градиентный бустинг (пример)

```
from sklearn.datasets import fetch_california_housing
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.metrics import mean_absolute_percentage_error
```

```
housing = fetch_california_housing()
X, y = housing.data, housing.target
X.shape, y.shape, housing.target_names

((20640, 8), (20640,), ['MedHouseVal'])
```

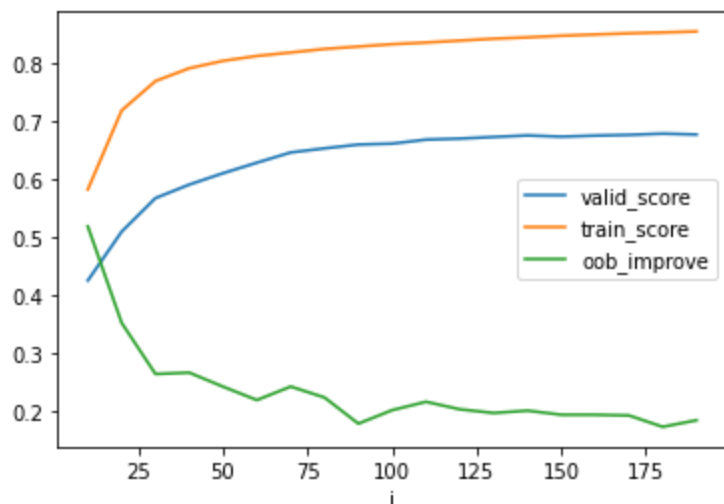
```
N = 15000
X_train, y_train = X[:N], y[:N]
X_test, y_test = X[N:], y[N:]
```

```
n_estimators = 100
boosting = GradientBoostingRegressor(n_estimators=n_estimators, learning_rate=0.1,
                                     max_depth=5,
                                     max_leaf_nodes=10,
                                     subsample=0.75, # stochastic if <1.0
                                     ccp_alpha=0.0, # pruning
                                     warm_start=True, # add trees to the existing forest
                                     random_state=0)

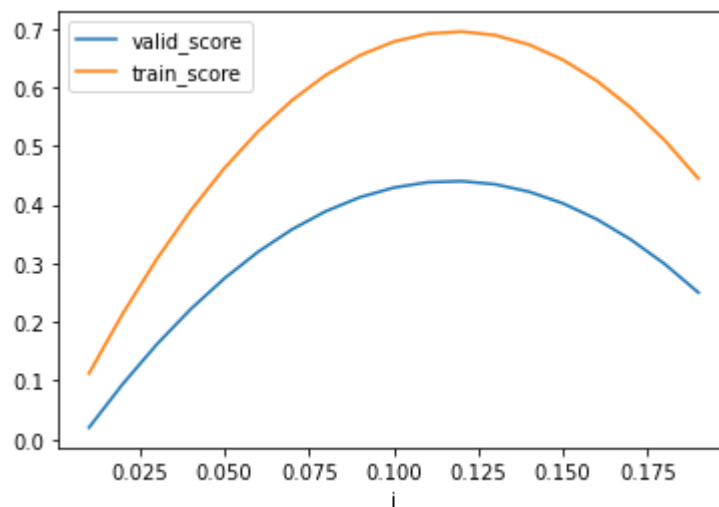
#boosting.fit(X_train, y_train)
history = sklearn_fit_history(boosting, n_estimators, X_train, y_train, (X_test, y_test))
```

Градиентный бустинг (пример)

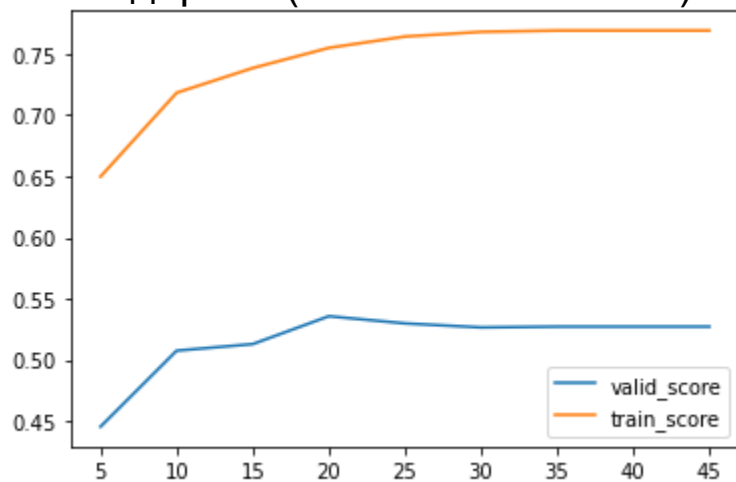
Качество от размера ансамбля



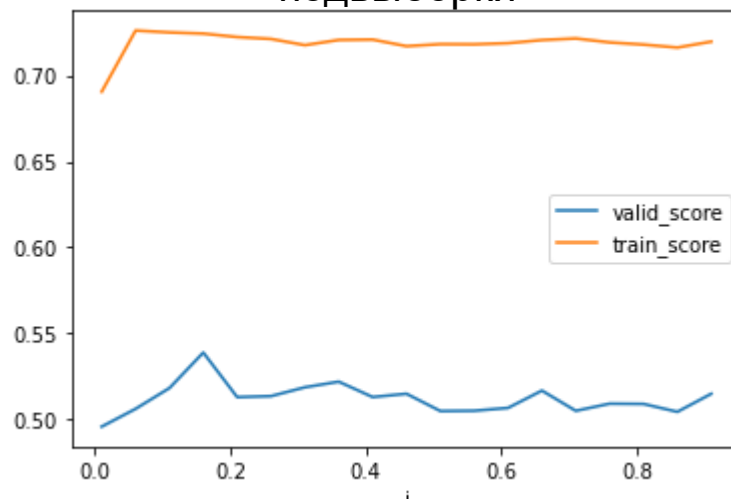
Качество от shrinkage



Качество от сложности дерева (max число листьев)



Качество от размера подвыборки

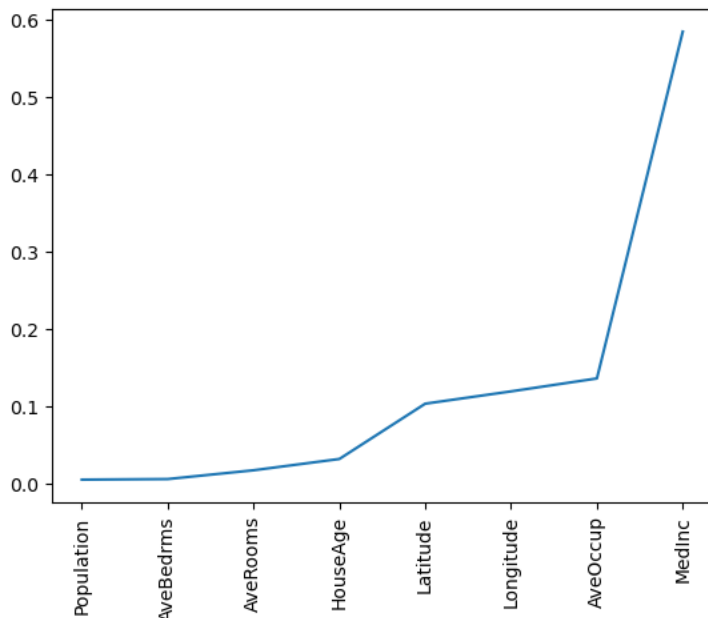


Градиентный бустинг (пример)

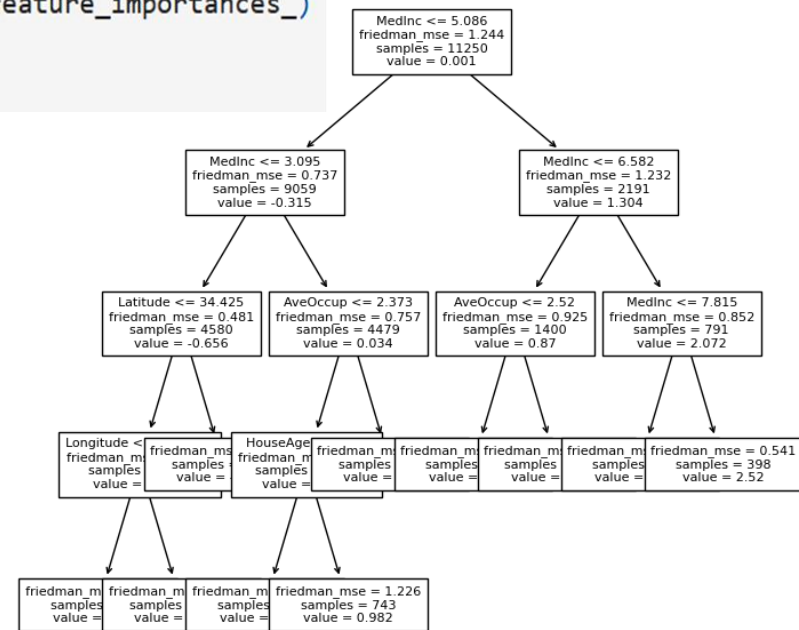
- Оценить важность переменных по ансамблю:

```
importance = pd.Series(index=housing.feature_names, data=boosting.feature_importances_)
importance.sort_values().plot()
plt.xticks(rotation='vertical');
```

средний прирост качества разбиения по переменной x_j всем деревьям ансамбля

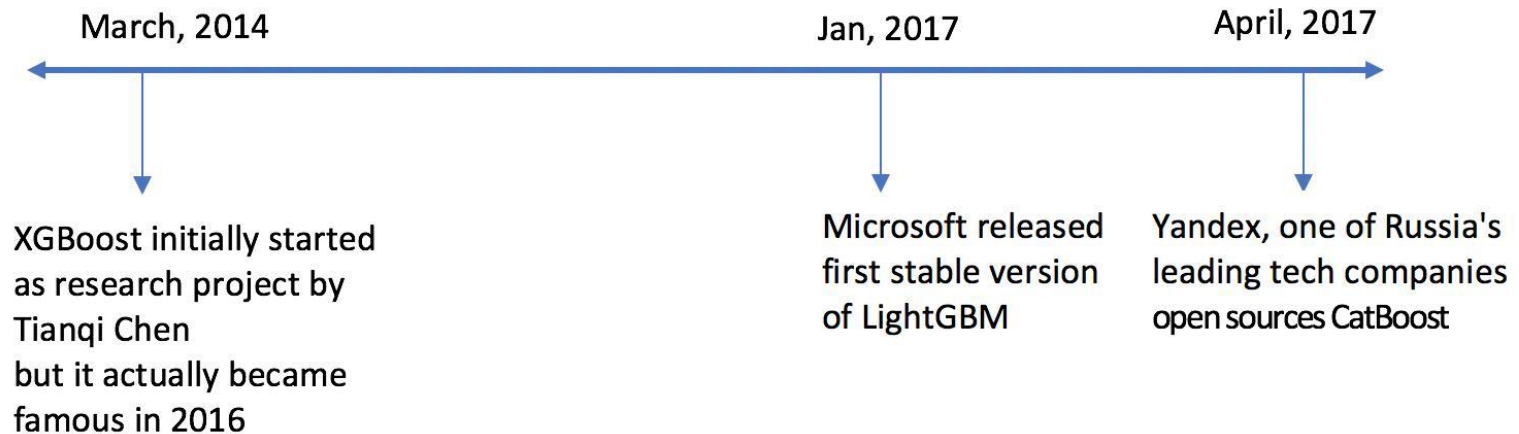


- Можно «добраться» до каждого дерева в ансамбле



```
tree = boosting.estimators_[0][0]
plot_tree(tree, fontsize=8, feature_names=housing.feature_names)
plt.gcf().set_size_inches(8, 8)
```

Современные алгоритмы бустинга

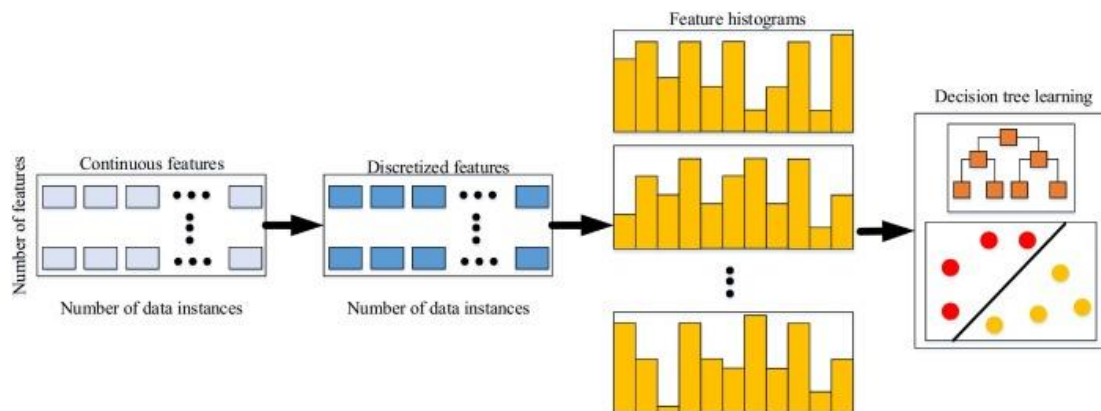


■ Общие особенности:

- Распараллеливание, поддержка GPU, TPU, возможности AutoML
- Возможность дообучения для больших объемов данных
- Разные стандартные и пользовательские метрики и функции потерь
- Бэггинг, подвыборки и случайные подпространства, могут быть как RF
- L_p регуляризации, ранняя остановка и контроль learning rate
- Гистограммные методы поиска разбиений для числовых признаков
- Колбэки и кастомное журналирование

Гистограммный подход для числовых признаков

- Предварительная дискретизация числовой переменной:
 - обычно на равные интервалы (buckets), получаем порядковую переменную с «весами» – число наблюдений в интервале



- число перебираемых вариантов разбиения – число интервалов, а не число различных значений; ускоряется расчет критериев разбиения (за счет весов интервалов, удаления пустых или слабо заполненных); рекурсивный «досчет» на интервалах
- эффективно сочетается с ростом дерева «в глубину» (list-wise)
- распараллеливается расчет гистограмм

Отличия (самые важные)

■ LightGBM:

- Оптимизирован под рост «в глубину»
- Gradient-based One-Side Sampling (GOSS) – адаптивный сэмплинг пропорционально важности примера (градиенту) при поиске разбиения
- Exclusive Feature Bundling – жадный алгоритм «группировки» значений категориальных признаков на непересекающиеся группы

■ CatBoost:

- Оптимизирован под ODT
- SWOE кодирование категориальных предикторов
- Упорядоченный семплинг (для борьбы с переобучением градиентов)

■ XGBoost:

- Оптимизирован под рост «в ширину»
- Ньютоновский бустинг - критерий поиска разбиений и/или обрубания, учитывающий качество и регуляризацию всего ансамбля

Дополнительные модификации деревьев решений и методов подвыборок бустинга

■ Цель:

- Ускорить процесс построения дерева, возможно за счет ухудшения качества и внесения дополнительной случайности («шума»)
- Это плохо для обычных деревьев и иногда для бэггинга (может увеличивать смещение базовых моделей), но хорошо для бустинга, т.к. он уменьшает не только дисперсию, но и смещение

■ «Ускорение»/ослабление обучения базовых деревьев решений:

- Предобработка (сокращение перебора): для категориальных переменных **SWOE** (отображаем на порядковую шкалу) и жадная **группировка** значений категориальных переменных; для числовых – гистограммный подход
- Уменьшение выборки при поиске разбиения – взвешенный **градиентный sampling, упорядоченный бустинг**
- Упрощения структуры деревьев: рост «в глубину» - **list-wise** (сложный) и в «ширину» **level-wise** (по уровням – простой), Oblivious Decision Trees (**ODT**) - решающие таблицы

Предобработка категориальных признаков

■ SWOE (по текущей подвыборке) для категориальных:

□ Пусть $\{v_1, \dots, v_k\}$ – множество значений категориальной переменной x в подвыборке Z_m

□ Для бинарного отклика $\rho_1 = P(y_i = 1 | (x_i, y_i) \in Z_m)$, c - параметр

$$x = v \Rightarrow SWOE_x(v) = \log \left(\frac{\sum_{x_i \in Z_m} I[x_i = v] I[y_i = 1] + c \rho_1}{\sum_{x_i \in Z_m} I[x_i = v] I[y_i = 0] + c(1 - \rho_1)} \right)$$

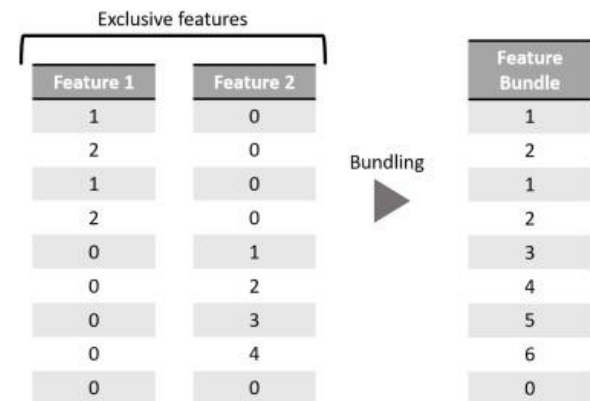
□ Для числового отклика $\rho = E(y_i | (x_i, y_i) \in Z_m)$, c – параметр

$$x = v \Rightarrow SWOE_x(v) = \frac{\sum_{x_i \in Z_m} y_i I[x_i = v] + c \rho}{\sum_{x_i \in Z_m} I[x_i = v] + c}$$

■ Жадная группировка значений категориальных переменных:

□ Кодировем по порядку часто встречающиеся комбинации значений признаков, а не их декартово произведение

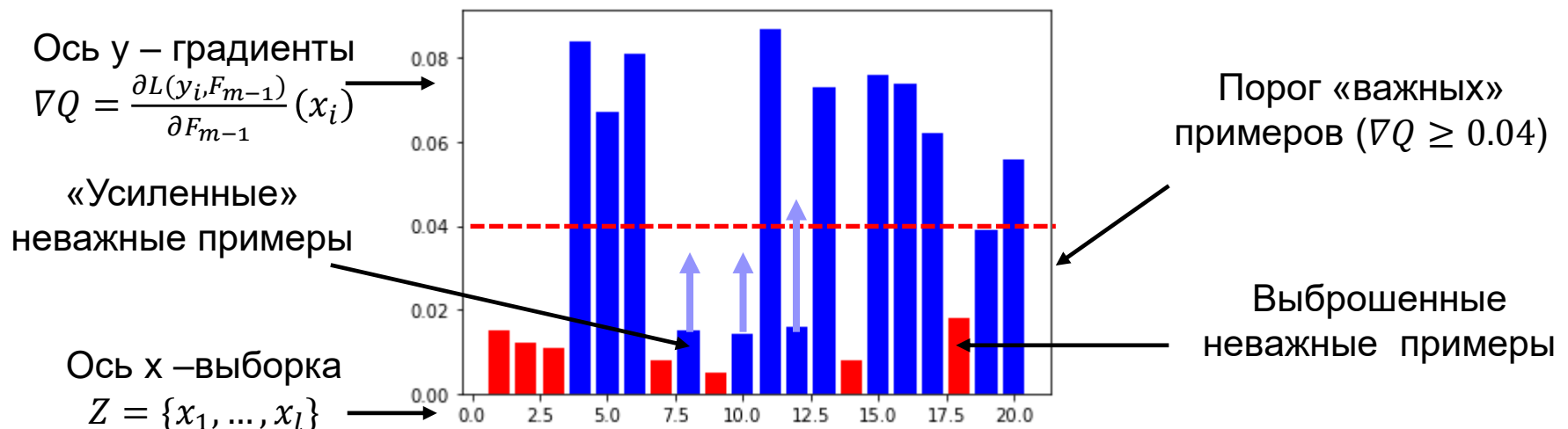
□ Эффективно при One-hot-encoding



Градиентный sampling

- Идея близка к Arc-x4:

- учить деревья (или искать отдельные разбиения) на подмножестве «важных» примеров
- важность примера – значение градиента на нем (псевдоостаток)
- популярный вариант – отбираем топ % «важных» примеров, их всегда берем, а среди «не важных», часть берем случайно, но увеличиваем их градиент (меняем вес) пропорционально числу отобранных, так, чтобы не поменялось распределение «важных».



LightGBM (пример)

```
from lightgbm import LGBMClassifier, early_stopping, record_evaluation, plot_importance
```

```
X_train, X_test, y_train, y_test = covtype_split
```

```
lgbm_classifier = LGBMClassifier(boosting_type="gbdt", # dart, rf
                                num_leaves=10,
                                max_depth=-1,
                                learning_rate=0.05,
                                min_child_samples=20, # min_samples_leaf
                                n_estimators=2000,
                                subsample=0.15, # subsample % for stochastic
                                subsample_freq=1, # how many times to subsample
                                colsample_bytree=0.15, # features by trees
                                class_weight="balanced",
                                reg_alpha=1.0 # l1 regularization
                                )
```

```
history = {}
lgbm_classifier.fit(X_train, y_train, feature_name=covtype.feature_names,
                   eval_set=[(X_test, y_test), (X_train, y_train)],
                   callbacks=[early_stopping(stopping_rounds=10), record_evaluation(history)])
```

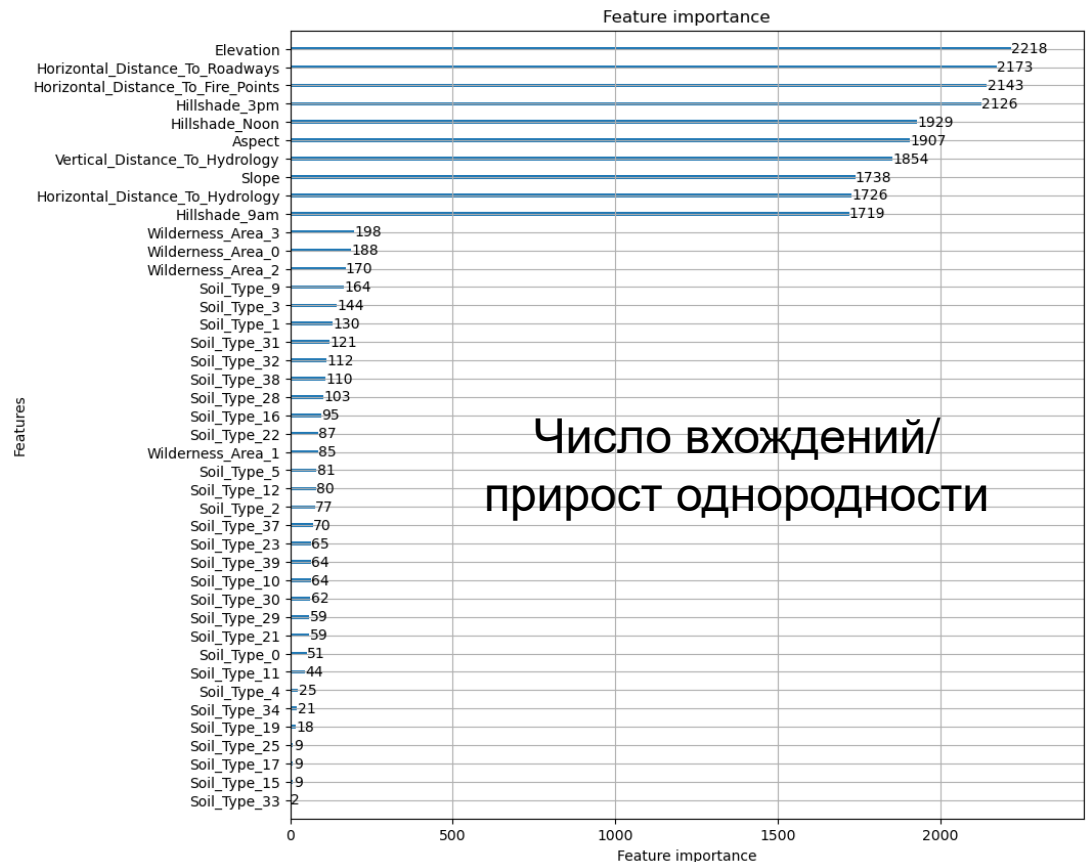
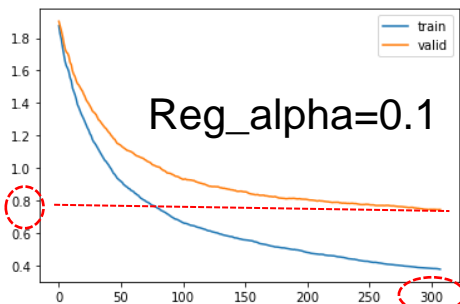
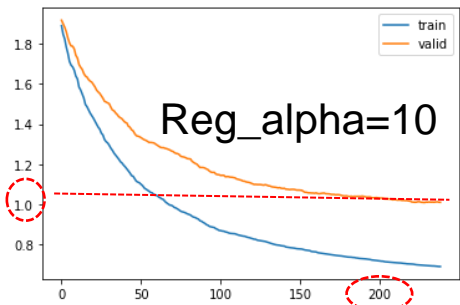
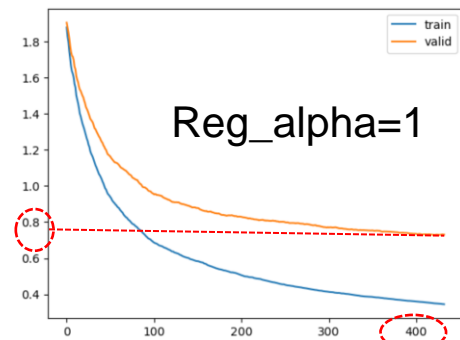
Training until validation scores don't improve for 10 rounds

Early stopping, best iteration is:

[423] training's multi_logloss: 0.349899 valid_0's multi_logloss: 0.728702

LightGBM (пример)

```
train = history["training"]["multi_logloss"]
valid = history["valid_0"]["multi_logloss"]
pd.DataFrame(dict(train=train, valid=valid)).plot()
```



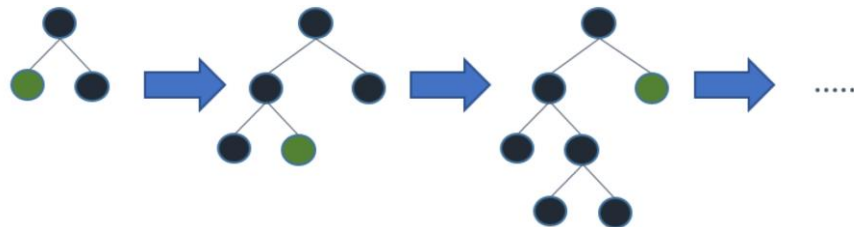
Число вхождений/
прирост однородности

```
plot_importance(lgbm_classifier, figsize=(10, 10));
```

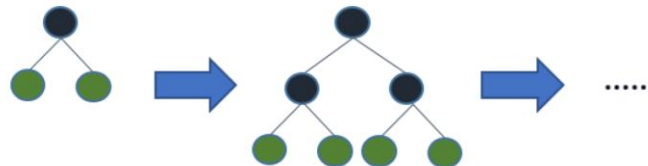
Упрощения роста дерева

■ Стратегии роста дерева:

- «**в глубину**» (list-wise) – классический, жадный, вычислительно долгий, сложно контролировать сложность всего дерева



- «**в ширину**» (level-wise) – упрощенный вариант, на каждом шаге плюс уровень для всех текущих листьев



- «**Небрежные**» деревья решений (Oblivious Decision Trees – ODT) или решающие таблицы – «в ширину», а еще правило разбиения (и переменная, и точка разбиения) одинаковые для всего уровня

Oblivious Decision Trees

- Основная идея – искать одно разбиение на весь уровень:

- $b_d(x)$ решающее правило (сплит) для всего уровня d

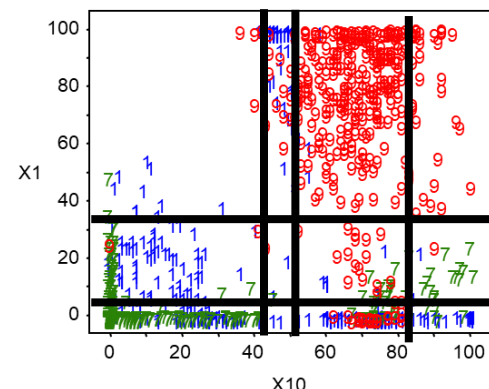
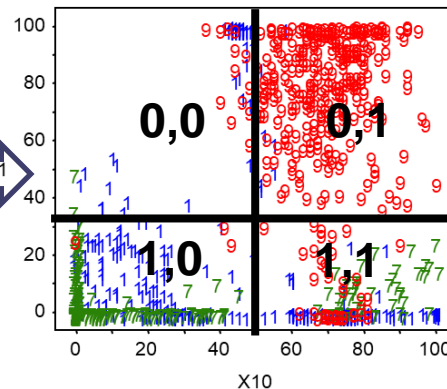
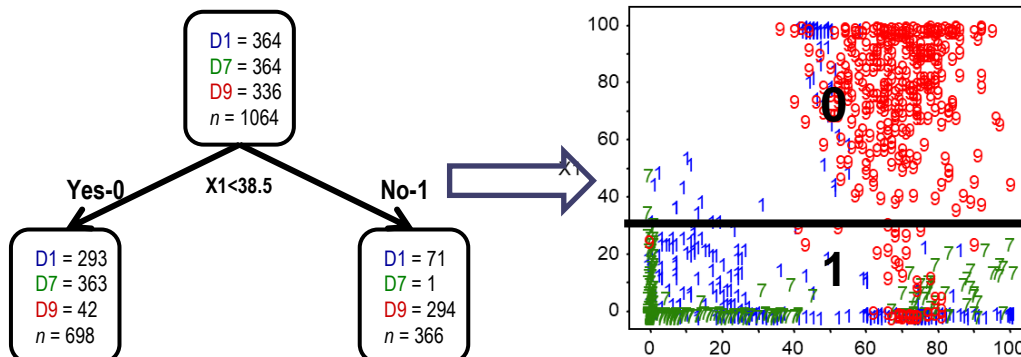
- Решающая таблица:

- Для дерева глубины D пространство X делится на 2^D ячеек с решающим правилом:

$$B: \{0,1\}^D \rightarrow Y,$$

$$a(x) = B(b_1(x), \dots, b_D(x))$$

$b_1(x), \dots, b_D(x)$ – вектор прогнозов определяет «координаты» ячейки



Алгоритм обучения (бинарного) ODT

■ Алгоритм разбиения по уровням (для уровня d)

- Сформировать множество *гипотез* $\{f_i\}$ для разбиения по всем признакам, $f_i: X \rightarrow \{0,1\}$ разбивает все пространство на два региона
- Рассчитать значение *критерия разбиения* для каждой гипотезы с учетом уже существующего разбиения уровня d и выбрать лучшую по критерию (например, по увеличению однородности):

$$\Delta i = \sum_{p \in leaves_d} \left(i_p - \frac{n_{p,0}i_{p,0} + n_{p,1}i_{p,1}}{n_p} \right) \rightarrow \max$$

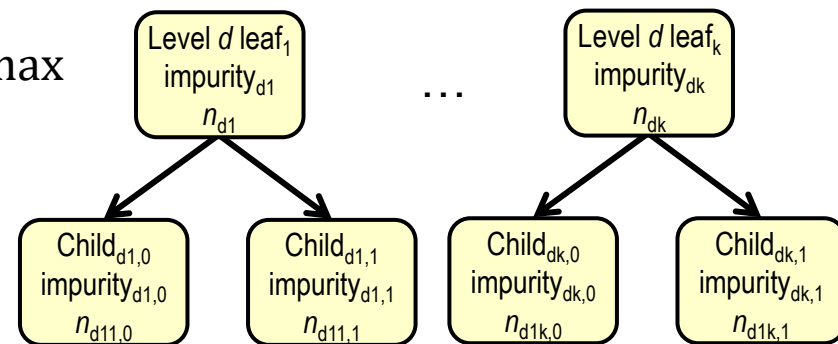
$leaves_d$ - листья на уровне d , их 2^d

i_p, n_p - однородность и число

примеров одного из листьев уровня d

i_{p*}, n_{p*} - однородность и число

примеров в дочерних узлах одного из листьев уровня d



- Дорастить дерево «в ширину»: каждый лист уровня d превращается во внутренний узел с двумя новыми ветвями

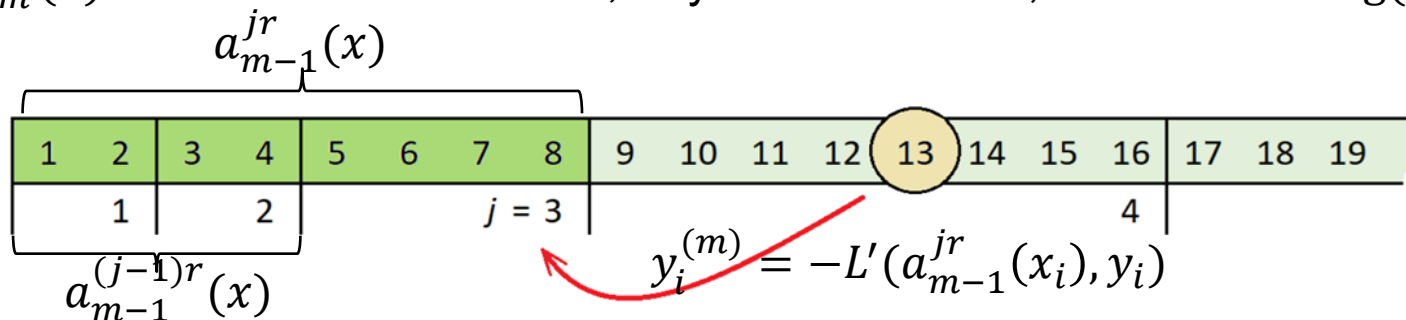
Упорядоченный бустинг

- Важная проблема градиентного бустинга – target leakage:
 - «**Переобучение градиента**» за счет того, что на шаге m градиенты функции потерь (псевдоостатки) считаются с учетом отклика и на тех же точках $\{(x_i)_{i=1}^l$, на которых обучался ансамбль a_{m-1} на предыдущем шаге
- Как этого избежать?
 - Считать градиент функции потерь для x_i по ансамблю с шага $a_{m-1}^{(i)}$, который бы учился на $Z_m^{(i)}$, таком что $x_i \notin Z_m^{(i)}$.
 - Но считать и хранить l ансамблей одновременно - плохая идея ...
- Основные идеи упорядоченного бустинга:
 - Цель - стараться вычислять градиент (псевдоостаток) для x_i по ансамблю $a_{m-1}^{(i)}$, **который не учился на x_i**
 - Строить обучающие подвыборки **последовательно** увеличивая размер (например, удваивая длину, тогда нужно не $O(l)$ моделей, а $O(\log(l))$)
 - Нужно несколько случайно **перемешанных** подвыборок, чтобы они не перекрывались

Генерация выборок для упорядоченного бустинга

■ Основная идея:

- Заимствована из онлайн обучения (дообучаемся как накопятся данные, например, удвоилась выборка), но еще с перестановками
- s_0, s_1, \dots, s_k - случайные упорядоченные перестановки выборки $\{x_i\}_{i=1}^l$
- X^{jr} - подвыборка первых j элементов из $s_r, r > 0$ - s_0 - «запасная»
- $g_m^{jr}(x_i) = -L'(a_{m-1}^{jr}(x_i), y_i)$ - i -я координата (в точке x_i) градиента функции потерь (псевдоостаток) модели, которая не обучалась на x_i
- если дообучаемся после удвоения, то $j = \text{int}(\log_2(i - 1))$ задает длину выборки, в которой еще не учился i -й объект
- $a_m^{jr}(x)$ - ансамбль-заготовка, обученный на X^{jr} , их всего $k * \log(l)$



Алгоритм упорядоченного бустинга (CatBoost)

■ Повторить $m = 1, \dots, M$ раз, где M – размер ансамбля:

1. Выбрать **случайно** перестановку $s_r \in \{s_1, \dots, s_k\}$, s_0 - отложена
2. Для всех наблюдений x_i , $i = 1, \dots, l$ вычислить **несмещенный антиградиент** (псевдоостаток), находя для каждого i такое максимальное j , чтобы a_{m-1}^{jr} не учился еще на x_i : $g_m^{jr}(x_i) = L'(a_{m-1}^{jr}(x_i), y_i)$
3. Обучить на найденных псевдоостатках и примерах $\{(x_i, -g_m^{jr}(x_i))\}$ новое **специальное базовое** дерево $b_m^{base}(x)$, где разбиения и прогнозы в листьях для каждого наблюдения считаются с учетом порядка X^{jr}
4. Для всех $r = 0, \dots, k$ берем **общую структуру дерева** $b_m^{base}(x)$ («жульничаем», чтобы не перестраивать дерево на всех перестановках), а прогноз отклика в листьях пересчитываем на всех s_r с учетом X^{jr}
5. Получаем, во-первых, **базовые модели** $b_m^{jr}(x)$ для ансамблей-заготовок $a_m^{jr}(x)$, и $b_m^0(x)$, пересчитанный на s_0 для добавления в **финальный** α_m^0
6. Для всех jr и 0 находим **градиентный шаг** α_m^* , где $*$ = jr или $*$ = 0:

$$\alpha_m^* = \underset{\alpha}{\operatorname{argmin}} \sum_{i=1}^N L(y_i, a_{m-1}^*(x_i) + \alpha b_m^*(x_i))$$

Алгоритм построения специальных базовых деревьев в CatBoost

Algorithm 2: Building a tree in CatBoost

input : $M, \{(\mathbf{x}_i, y_i)\}_{i=1}^n, \alpha, L, \{\sigma_i\}_{i=1}^s, Mode$

$grad \leftarrow CalcGradient(L, M, y);$
 $r \leftarrow random(1, s);$
if $Mode = Plain$ **then**
 $G \leftarrow (grad_r(i) \text{ for } i = 1..n);$
if $Mode = Ordered$ **then**
 $G \leftarrow (grad_{r, \sigma_r(i)-1}(i) \text{ for } i = 1..n);$
 $T \leftarrow \text{empty tree};$
foreach *step of top-down procedure* **do**
 foreach *candidate split* c **do**
 $T_c \leftarrow \text{add split } c \text{ to } T;$
 if $Mode = Plain$ **then**
 $\Delta(i) \leftarrow avg(grad_r(p) \text{ for } p : leaf_r(p) = leaf_r(i)) \text{ for } i = 1..n;$
 if $Mode = Ordered$ **then**
 $\Delta(i) \leftarrow avg(grad_{r, \sigma_r(i)-1}(p) \text{ for } p : leaf_r(p) = leaf_r(i), \sigma_r(p) < \sigma_r(i)) \text{ for } i = 1..n;$
 $loss(T_c) \leftarrow cos(\Delta, G)$
 $T \leftarrow argmin_{T_c}(loss(T_c))$
if $Mode = Plain$ **then**
 $M_{r'}(i) \leftarrow M_{r'}(i) - \alpha avg(grad_{r'}(p) \text{ for } p : leaf_{r'}(p) = leaf_{r'}(i)) \text{ for } r' = 1..s, i = 1..n;$
if $Mode = Ordered$ **then**
 $M_{r', j}(i) \leftarrow M_{r', j}(i) - \alpha avg(grad_{r', j}(p) \text{ for } p : leaf_{r'}(p) = leaf_{r'}(i), \sigma_{r'}(p) \leq j) \text{ for } r' = 1..s, i = 1..n, j \geq \sigma_{r'}(i) - 1;$
return T, M

■ Особенности:

- «прогноз» в листьях (а значит и всего дерева) не константа для всех примеров листа, а вектор длины l с усреднением прогноза для x_i с учетом его листа и порядка в X^{jr}
- Потери (критерий оценки разбиения) – косинусная мера сходства с вектором несмещенных антиградиентов
- Пересчет «векторного» прогноза в «точечный» – усреднением по всем

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

$T_t, \{M_r\}_{r=1}^s \leftarrow BuildTree(\{M_r\}_{r=1}^s, \{(\mathbf{x}_i, y_i)\}_{i=1}^n, \alpha, L, \{\sigma_i\}_{i=1}^s, Mode);$
 $leaf_0(i) \leftarrow GetLeaf(\mathbf{x}_i, T_t, \sigma_0) \text{ for } i = 1..n;$
 $grad_0 \leftarrow CalcGradient(L, M_0, y);$
 foreach *leaf* j **in** T_t **do**
 $b_j^t \leftarrow -avg(grad_0(i) \text{ for } i : leaf_0(i) = j);$
 $M_0(i) \leftarrow M_0(i) + \alpha b_{leaf_0(i)}^t \text{ for } i = 1..n;$

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

Пример CatBoost с ordered boosting

```
model = CatBoostClassifier(iterations=500, # Number of boosting iterations
                           learning_rate=0.3, # Learning rate
                           #grow_policy='Depthwise',
                           depth=5, # Depth of the tree
                           verbose=100, # Print training progress every 50 iterations
                           early_stopping_rounds=10, # stops training if no improvement in 10 consecutive rounds
                           loss_function='MultiClass') # used for Multiclass classification tasks
```

0:	learn: 1.3244454	test: 1.3274746 best: 1.3274746 (0)	total: 10.8ms	remaining: 5.39s
100:	learn: 0.4770844	test: 0.5501835 best: 0.5501835 (100)	total: 1.25s	remaining: 4.95s
200:	learn: 0.3820112	test: 0.5085822 best: 0.5085822 (200)	total: 2.53s	remaining: 3.76s
300:	learn: 0.3180690	test: 0.4868846 best: 0.4868102 (299)	total: 3.6s	remaining: 2.38s
400:	learn: 0.2719426	test: 0.4756932 best: 0.4756932 (400)	total: 4.96s	remaining: 1.23s
499:	learn: 0.2394187	test: 0.4701613 best: 0.4699182 (495)	total: 6.32s	remaining: 0us

```
bestTest = 0.4699181794
bestIteration = 495
```

Shrink model to first 496 iterations.

0:	learn: 1.2894816	test: 1.2931179 best: 1.2931179 (0)	total: 9.73ms	remaining: 4.85s
100:	learn: 0.4119708	test: 0.5314217 best: 0.5314217 (100)	total: 897ms	remaining: 3.54s
200:	learn: 0.2999559	test: 0.4992096 best: 0.4992096 (200)	total: 1.8s	remaining: 2.68s
300:	learn: 0.2308821	test: 0.4853332 best: 0.4853332 (300)	total: 2.71s	remaining: 1.79s

Stopped by overfitting detector (10 iterations wait)

```
bestTest = 0.4808030753
bestIteration = 355
```

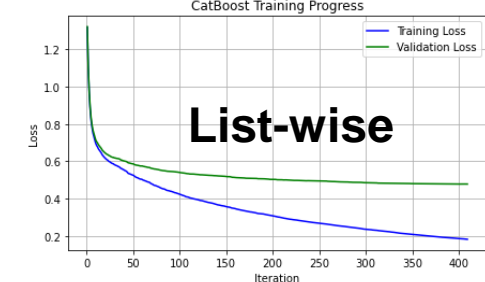
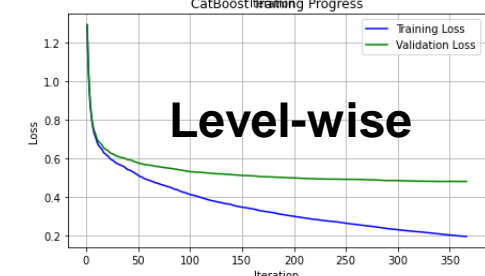
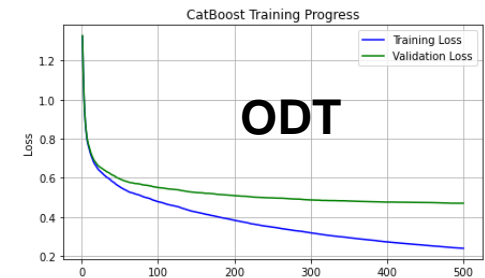
Shrink model to first 356 iterations.

0:	learn: 1.3175380	test: 1.3213286 best: 1.3213286 (0)	total: 11.4ms	remaining: 5.71s
100:	learn: 0.4224715	test: 0.5395996 best: 0.5395996 (100)	total: 1.02s	remaining: 4.02s
200:	learn: 0.3073328	test: 0.5033971 best: 0.5032791 (197)	total: 2.08s	remaining: 3.1s
300:	learn: 0.2355510	test: 0.4854600 best: 0.4854600 (300)	total: 3.18s	remaining: 2.1s
400:	learn: 0.1859574	test: 0.4783043 best: 0.4780336 (398)	total: 4.29s	remaining: 1.06s

Stopped by overfitting detector (10 iterations wait)

```
bestTest = 0.4780336062
bestIteration = 398
```

Shrink model to first 399 iterations.



Сравнение CatBoost с упорядоченным и обычным boosting

```
model = CatBoostClassifier(iterations=500, # Number of boosting iterations
                           learning_rate=0.3, # Learning rate
                           depth=3, # Depth of the tree
                           boosting_type = "Plain",
                           verbose=100, # Print training progress every 50 iterations
                           early_stopping_rounds=10, # stops training if no improvement in 10 consecutive rounds
                           loss_function='MultiClass') # used for Multiclass classification tasks
```

0:	learn: 1.3214366	test: 1.3220541	best: 1.3220541 (0)	total: 6.69ms	remaining: 3.34s
100:	learn: 0.5648893	test: 0.5993844	best: 0.5993844 (100)	total: 793ms	remaining: 3.13s
200:	learn: 0.5041605	test: 0.5639954	best: 0.5639954 (200)	total: 1.63s	remaining: 2.42s
300:	learn: 0.4606368	test: 0.5450221	best: 0.5450221 (300)	total: 2.57s	remaining: 1.7s
400:	learn: 0.4276630	test: 0.5317555	best: 0.5317555 (400)	total: 3.51s	remaining: 867ms
499:	learn: 0.4005115	test: 0.5217715	best: 0.5215625 (496)	total: 4.45s	remaining: 0us

bestTest = 0.5215625037
bestIteration = 496

Shrink model to first 497 iterations.

Обычный – быстрее делает итерации,
но хуже сходится

0:	learn: 1.3244454	test: 1.3274746	best: 1.3274746 (0)	total: 10.8ms	remaining: 5.39s
100:	learn: 0.4770844	test: 0.5501835	best: 0.5501835 (100)	total: 1.25s	remaining: 4.95s
200:	learn: 0.3820112	test: 0.5085822	best: 0.5085822 (200)	total: 2.53s	remaining: 3.76s
300:	learn: 0.3180690	test: 0.4868846	best: 0.4868102 (299)	total: 3.6s	remaining: 2.38s
400:	learn: 0.2719426	test: 0.4756932	best: 0.4756932 (400)	total: 4.96s	remaining: 1.23s
499:	learn: 0.2394187	test: 0.4701613	best: 0.4699182 (495)	total: 6.32s	remaining: 0us

bestTest = 0.4699181794
bestIteration = 495

Shrink model to first 496 iterations.

Упорядоченный

Особенности классического градиентного бустинга деревьев решений

- В сравнении с основным конкурентом случайным лесом:
 - уменьшает не только разброс, но и **смещение** всего ансамбля
 - с ростом числа базовых моделей не склонен к **переобучению**, когда нет шума (выбросов), но склонен когда **выбросы** есть
 - **плохо распараллеливается** (только на уровне отдельных моделей) и требует **больше вычислений** (пересчет псевдоостатков, поиск веса базовой модели на каждом шаге)
 - помимо ограничений на сложность есть и **регуляризация** (shrinkage-сокращение, штрафы L0, L1, L2, ранняя остановка)
 - может использовать идеи из случайного леса: **случайные подпространства** признаков при поиске разбиения (часто полезно) и дополнительно **бутстрепинг** в стохастическом градиентном бустинге (часто бесполезно)
 - базовые деревья (регионы и прогнозы в них) строятся **без использовании информации** о всем ансамбле – это плохо! А можно ли исправить? **ДА!**

Учет потерь ансамбля в каждом дереве

- Бустинг деревьев решений:

- Ансамбль: $F_m(x) = F_0 + \alpha_1 T_1(x) + \alpha_2 T_2(x) + \dots + \alpha_m T_m(x)$

$$F_m(x) = F_0 + \alpha_1 * \begin{array}{|c|c|} \hline \gamma_{R_{11}} & \dots \\ \hline \gamma_{R_{12}} & \gamma_{R_{13}} \\ \hline \end{array} + \alpha_2 * \begin{array}{|c|c|} \hline \gamma_{R_{22}} & \gamma_{R_{23}} \\ \hline \gamma_{R_{21}} & \gamma_{R_{24}} \\ \hline \end{array} + \dots + \alpha_m * \begin{array}{|c|c|} \hline \gamma_{R_{m2}} & \gamma_{R_{m1}} \\ \hline \gamma_{m3} & \end{array}$$

- Базовая модель – дерево $T_m(x) = \sum_{R \in R_m} \gamma_R I[x \in R]$, обученное на псевдоостатках в качестве вектора отклика $-\left[\frac{\partial L(y_i, F_{m-1})}{\partial F_{m-1}}(x_i)\right]_{i=1}^l$

- Ансамбль минимизирует потери: $Q_m = \sum_i L(y_i, F_m(x_i))$, а можно сразу искать $(\{\gamma_R\}_{R \in R_m}, R_m) = \operatorname{argmin}_{R_m, \gamma} Q_m$ при фиксированном Q_{m-1} ?

НЬЮТОНОВСКИЙ БУСТИНГ (XGBoost)

- Раскладываем потери в ряд Тейлора до 2 слагаемого:

$$\begin{aligned} & \sum_i L(y_i, F_{m-1}(x_i) + b(x_i)) \approx \\ & \approx \sum_i L(y_i, F_{m-1}(x_i)) + b(x_i) \frac{\partial L(y_i, F_{m-1})}{\partial F_{m-1}}(x_i) + \frac{1}{2} b^2(x_i) \frac{\partial^2 L(y_i, F_{m-1})}{(\partial F_{m-1})^2}(x_i) \dots \end{aligned}$$

- Обозначим:

$$b_i = b(x_i), g_i = \frac{\partial L(y_i, F_{m-1})}{\partial F_{m-1}}(x_i), h_i = \frac{\partial^2 L(y_i, F_{m-1})}{(\partial F_{m-1})^2}(x_i)$$

- Тогда приближенно потери:

$$\sum_i L(y_i, F_m(x_i)) \sim \sum_i [g_i b_i + \frac{1}{2} h_i b_i^2] + const$$

- Добавим регуляризацию L_2 на отклик и L_0 (число листьев) на сложность дерева, получим критерий для минимизации:

$$Q_m = \sum_i [g_i b_i + \frac{1}{2} h_i b_i^2] + \lambda_2 |T_m(x)| + \frac{1}{2} \lambda_1 \sum_{R \in R_m} \gamma_R^2 \rightarrow \min_{R_m, \gamma_R}$$

Ньютоновский бустинг (XGBoost)

- Логика вывода основных формул:

- Если зафиксируем структуру дерева (регионы R_m), то из $\frac{\partial Q_m}{\partial \gamma_R} = 0 \Rightarrow$

$$\gamma_R = \frac{\sum_{x_i \in R} g_i}{\lambda_1 + \sum_{x_i \in R} h_i}$$

- Подставляя γ_R в Q_m получим новый критерия поиска разбиения:

$$\Phi_m = -\frac{1}{2} \sum_{R \in R_m} \frac{\left(\sum_{x_i \in R} g_i\right)^2}{\lambda_1 + \sum_{x_i \in R} h_i} + \lambda_2 |T_m| \rightarrow \min$$

- Прирост для поиска бинарного разбиения:

$$Gain = -\frac{1}{2} \left[\frac{\left(\sum_{x_i \in R_{left}} g_i\right)^2}{\lambda_1 + \sum_{x_i \in R_{left}} h_i} + \frac{\left(\sum_{x_i \in R_{right}} g_i\right)^2}{\lambda_1 + \sum_{x_i \in R_{right}} h_i} - \frac{\left(\sum_{x_i \in R_{parent}} g_i\right)^2}{\lambda_1 + \sum_{x_i \in R_{parent}} h_i} \right] - \lambda_2$$

XGBoost

```
import xgboost as xgb
```

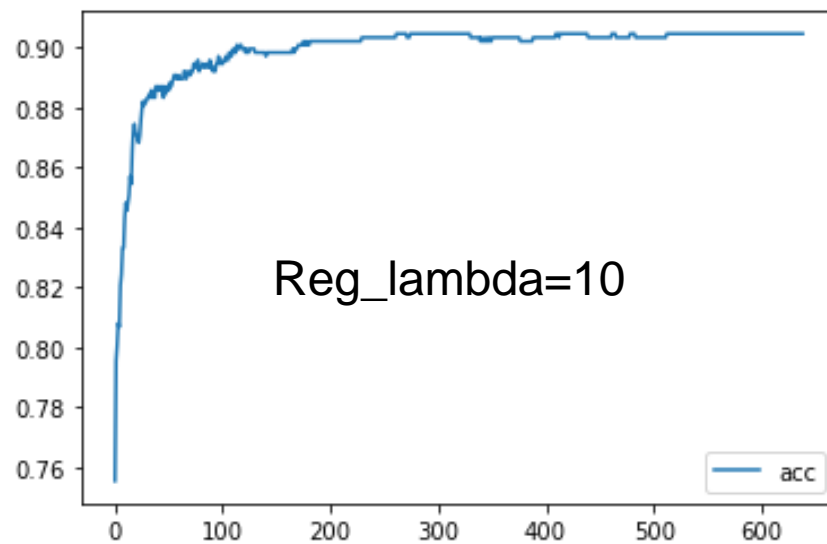
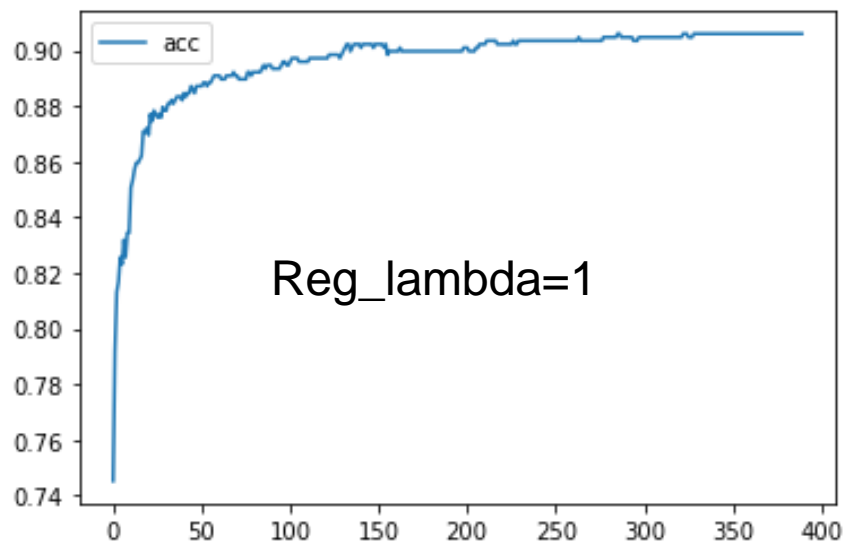
```
# https://xgboost.readthedocs.io/en/stable/parameter.html
param = dict(objective="multi:softprob", num_class=10, # softmax
              booster="gbtree", learning_rate=0.1, max_depth=5, subsample=0.5,
              alpha=5, # l1 regularization
              tree_method="auto") # approx, hist, gpu_hist
```

```
N = 1000
dtrain = xgb.DMatrix(digits.data[:N], label=digits.target[:N])
dvalid = xgb.DMatrix(digits.data[N:], label=digits.target[N:])
evallist = [(dtrain, 'train'), (dvalid, 'eval')]
```

```
xgb_classifier = xgb.train(param, dtrain, 1000, evals=evallist, early_stopping_rounds=10)
# xgb_classifier.save_model('mymodel')
```

```
[0]      train-mlogloss:2.03753   eval-mlogloss:2.08882
[1]      train-mlogloss:1.82993   eval-mlogloss:1.90860
[2]      train-mlogloss:1.66188   eval-mlogloss:1.76258
[3]      train-mlogloss:1.53163   eval-mlogloss:1.65200
...
[475]    train-mlogloss:0.16262   eval-mlogloss:0.45679
[476]    train-mlogloss:0.16262   eval-mlogloss:0.45679
```

XGBoost



```
result = []
for i in range(xgb_classifier.best_iteration):
    true = dvalid.get_label()
    pred = xgb_classifier.predict(dvalid, iteration_range=(0, i + 1))
    result.append(dict(acc=accuracy_score(true, np.argmax(pred, axis=-1))))
pd.DataFrame(result).plot()
```

Сравнение (по logloss) XGBoost, LGBM и CatBoost

	CatBoost		LightGBM		XGBoost	
	Tuned	Default	Tuned	Default	Tuned	Default
Adult	0.26974	0.27298 +1.21%	0.27602 +2.33%	0.28716 +6.46%	0.27542 +2.11%	0.28009 +3.84%
Amazon	0.13772	0.13811 +0.29%	0.16360 +18.80%	0.16716 +21.38%	0.16327 +18.56%	0.16536 +20.07%
Click prediction	0.39090	0.39112 +0.06%	0.39633 +1.39%	0.39749 +1.69%	0.39624 +1.37%	0.39764 +1.73%
KDD appetency	0.07151	0.07138 -0.19%	0.07179 +0.40%	0.07482 +4.63%	0.07176 +0.35%	0.07466 +4.41%
KDD churn	0.23129	0.23193 +0.28%	0.23205 +0.33%	0.23565 +1.89%	0.23312 +0.80%	0.23369 +1.04%
KDD internet	0.20875	0.22021 +5.49%	0.22315 +6.90%	0.23627 +13.19%	0.22532 +7.94%	0.23468 +12.43%
KDD upselling	0.16613	0.16674 +0.37%	0.16682 +0.42%	0.17107 +2.98%	0.16632 +0.12%	0.16873 +1.57%
KDD 98	0.19467	0.19479 +0.07%	0.19576 +0.56%	0.19837 +1.91%	0.19568 +0.52%	0.19795 +1.69%
Kick prediction	0.28479	0.28491 +0.05%	0.29566 +3.82%	0.29877 +4.91%	0.29465 +3.47%	0.29816 +4.70%

Epsilon dataset



Higgs dataset

	CatBoost	XGBoost	LightGBM
CPU (Xeon E5-2660v4)	527 sec	4339 sec	1146 sec
GTX 1080Ti (11GB)	18 sec	890 sec	110 sec

Dataset Epsilon (400K samples, 2000 features). Parameters: 128 bins, 64 leafs, 400 iterations.



Epsilon dataset

Higgs dataset

	CatBoost	XGBoost	LightGBM
CPU (Xeon E5-2660v4)	770 sec	881 sec	438 sec
GTX 1080Ti (11GB)	32 sec	91 sec	94 sec

Dataset Higgs (4M samples, 28 features). Parameters: 128 bins, 64 leafs, 400 iterations.

Смесь экспертов

■ Постановка:

- $a(x) = \sum g_m(x)b_m(x)$
- где $g_m: X \rightarrow \mathbb{R}^+$ - функция **компетентности**, строится (обучается) отдельно и зависит от x , нормирована $\sum g_m(x) = 1$
- можно нормировать через параметрическую softmax (получим голосование при $\gamma \rightarrow \infty$):

$$g_m(x) = \text{softmax}(\tilde{g}_1(x), \dots, \tilde{g}_M(x); \gamma) = \frac{e^{\gamma \tilde{g}_m(x)}}{\sum_{j=1}^M e^{\gamma \tilde{g}_j(x)}}$$

■ Виды функций компетентности (похоже на функции активации в нейросетях), экспертная или параметрическая привязка к:

- j -му $f_j(x)$ признаку $g(x) = 1/(1 + \exp(-\alpha f_j(x) - \beta))$
- направлению α – вектор $g(x) = 1/(1 + \exp(-x^T \alpha - \beta))$
- наблюдению α – вектор (наблюдение) $g(x) = \exp(-\beta(x - \alpha)^2)$
- α, β – могут обучаться вместе с ансамблем, обучаться заранее и фиксироваться или задаваться экспертом и фиксироваться

Выпуклая функция потерь

- Для выпуклых $L(a(x), y)$ и $\sum g_m(x) = 1$:

- выполняется неравенство Йенсена для эмпирического риска:

$$\begin{aligned} Q(a) &= \sum_{i=1}^l L\left(\left[\sum_{m=1}^M g_m(x_i) b_m(x_i)\right], y_i\right) \leq \\ &\leq \sum_{m=1}^M \left(\sum_{i=1}^l g_m(x_i) L(b_m(x_i), y_i) \right) = \sum_{m=1}^M (Q_m(g_m, b_m)) \rightarrow \min \end{aligned}$$

- Итерационный ЕМ алгоритм (в цикле до сходимости):

- Начальное приближение (случайные, равные, подобранные) g_1, \dots, g_M
- **М-шаг**: для всех $m = 1, \dots, M$ фиксируем g_m и находим

$$b_m = \operatorname{argmin}_b \sum_{i=1}^l g_m(x_i) L(b(x_i), y_i)$$

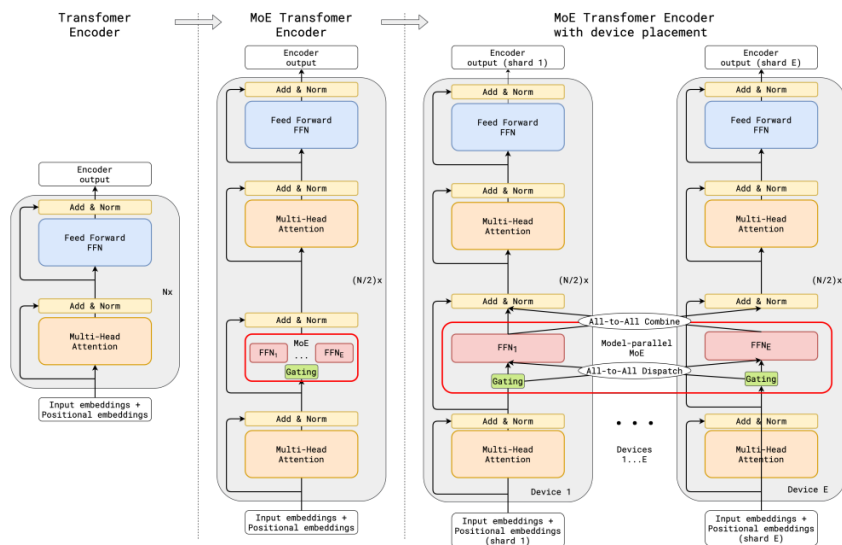
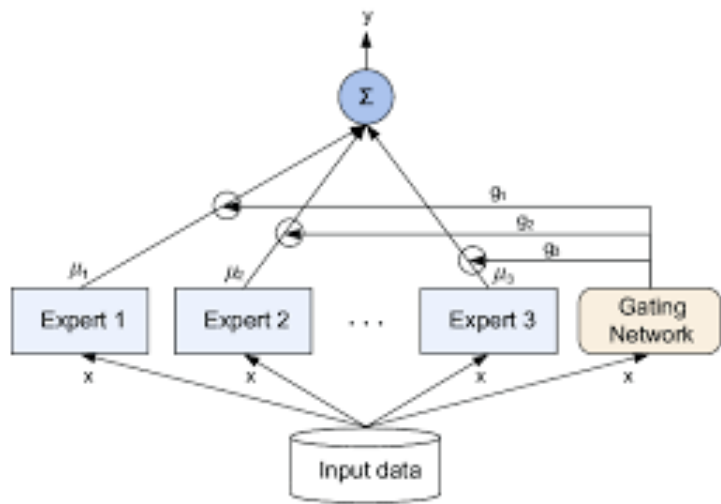
- **Е-шаг**: для всех $m = 1, \dots, M$ фиксируем b_m и находим

$$(\tilde{g}_1, \dots, \tilde{g}_M) = \operatorname{argmin}_{g_1, \dots, g_M} \sum_{i=1}^l L\left(\left[\sum_{m=1}^M g_m(x_i) b_m(x_i)\right], y_i\right)$$

- Нормировка для всех $m = 1, \dots, M$: $g_m(x) = \operatorname{softmax}(\tilde{g}_1(x), \dots, \tilde{g}_M(x); \gamma)$
- Если не стабилизировались g_m и b_m то переход на М-шаг

Продвинутые смеси экспертов

- Предложено много расширений и подходов к обучению:
 - Иерархические: $g_{i|j}(x)g_j(x)$
 - Байесовские: $g_m = P(b_m|Z) \sim P(b_m) \cdot P(Z|b_m)$ – «байесовский вес» базовой модели или «условная компетентность» на наборе Z , $P(Z|b_m)$ – правдоподобие модели компетентности
 - Нейросетевые (в том числе с глубоким обучением и трансформерами)

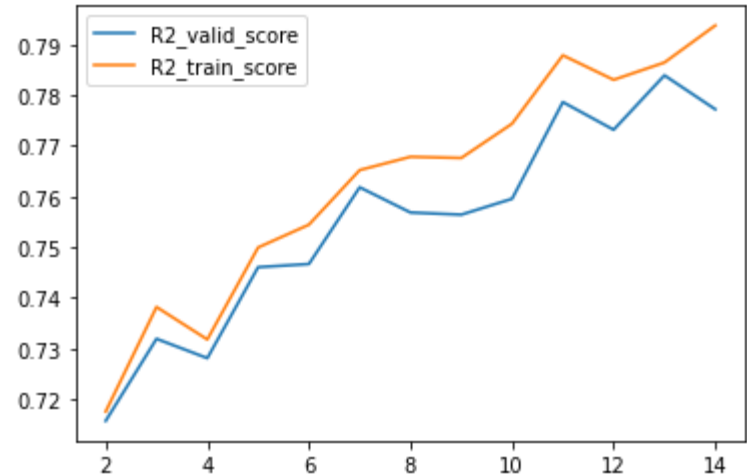


Пример МОЕ

- Смесь экспертов - МНК линейных и полиномиальных регрессий на наборе California Housing, оценка качества по R^2

```
import numpy as np
from smt.applications import MOE
housing = fetch_california_housing()
X, y = housing.data, housing.target
X.shape, y.shape, housing.target_names
N = 15000
X_train, y_train = X[:N], y[:N]
X_test, y_test = X[N:], y[N:]
def sklearn_fit_history_moe(X_train, y_train, X_valid, y_valid):
    result = []
    for i in range(2, 15, 1):
        print(f"Number of experts={i}")
        moe = MOE(n_clusters=i, allow=["LS", "QP"])
        moe.set_training_values(X_train, y_train)
        moe.train()
        d = {}
        d["i"] = i
        y_moe = moe.predict_values(X_valid)
        d["R2_valid_score"] = r2_score(y_valid, y_moe)
        y_moe = moe.predict_values(X_train)
        d["R2_train_score"] = r2_score(y_train, y_moe)
        result.append(d)
    return pd.DataFrame(data=result).set_index("i")
```

```
history=sklearn_fit_history_moe(X,y,X_test, y_test)
history.plot()
```



Number of experts=2

LS 29.92791962496508
QP 27.18807751579454
Best expert = QP
LS 5.501784342614732
QP 11.215110336591684
Best expert = LS

Number of experts=10

LS 10.29898080711618
QP 9.687474569666911
Best expert = QP
LS 10.528101535305112
QP 9.575876793498624
Best expert = QP
LS 5.224169041577111
QP 4.791085016027925
Best expert = QP
LS 2.1439357658079537e-05
QP 2.080389114006695e-05
Best expert = QP
LS 7.939421703962029
QP 68.86759267628352
Best expert = LS

LS 3.1637648277425874
QP 2.9549016444130336
Best expert = QP
LS 4.162144513803613
QP 3.9500173244680123
Best expert = QP
LS 7.296109359634265
QP 6.289615840759818
Best expert = QP
LS 6.771505099293365
QP 6.377682524464259
Best expert = QP
LS 7.732637050336106
QP 8.768475340330385
Best expert = LS

Выводы по ансамблям

- Позволяют существенно повышать качество базовых моделей
- Базовые модели часто это деревья решений (универсальная, не очень точная, не стабильная модель, что хорошо)
- Ансамбли бывают разных типов для разных задач
- Модели «из коробки» - Random Forest и градиентный бустинг
- ЕСОС, смеси экспертов и stacking тоже важны для своих задач
- Управлять качеством ансамбля можно варьируя сложность ансамбля (размер и/или гибкость агрегационной функции), сложность базовой модели и случайность (зависит от подвыборок и настроек базовых моделей)
- Большинство ансамблей уменьшают дисперсию прогноза, но некоторые могут уменьшать и смещение
- Основной минус – долго строить и применять, как правило теряется интерпретация исходных базовых моделей