Градиентный Бустинг

Задача безусловной оптимизации

Чтобы найти $x^* = \underset{x \in \mathbb{R}^n}{\operatorname{argmin}} f(x)$ необходимо решить $\nabla f(x) = 0$

Если $f(x)$ выпуклая $\Rightarrow$ единственный минимум

Если $f(x)$ невыпуклая:

- решения может не быть (функция вогнута, не имеет минимума)
- решений может быть много $\Rightarrow$ необходимо определить, является ли оно экстремумом, если да, то максимум или минимум.

Если нет аналитического решения $\Rightarrow$ **итерационная процедура**, определяющее решение приближенно: $|f(x) - f(x*)| \leq \varepsilon$

Поиск приближенного решения

Метод решения задачи оптимизации – построение приближения к решению исходной задачи (точке минимума *x**) на основе информации о характеристиках целевой функции.

- *Методы нулевого порядка* используют только значения функции
- *Методы первого порядка* – первые производные
- *Методы второго порядка* – вторые производные

$$y = f(\mathbf{x} + \Delta\mathbf{x}) \approx f(\mathbf{x}) + J(\mathbf{x})\Delta\mathbf{x} + \frac{1}{2}\Delta\mathbf{x}^{\mathrm{T}} H(\mathbf{x})\Delta\mathbf{x}$$

Общий вид метода оптимизации

Многие методы оптимизации имеют вид

$$x^{k+1} = x^k + \eta_k d^k, d^k \in \mathbb{R}^n, \eta_k \in \mathbb{R}, k = 0,1 \ldots$$

x^0 - начальное приближение

d^k – направление минимизации, определяется характеристиками минимизируемой функции и выбранной процедуры

η_k – длина шага (в МО называют скорость обучения)

- Если точное решение находится за конечное число шагов, метод называется *конечношаговым,* иначе – *бесконечношаговым*
- Бесконечношаговый сходится, если $x^k \to x^*, k \to \infty$
- Скорость сходимости (число шагов, не количество вычислений):
  - Линейная $\left\|x^{k+1} - x^*\right\| \le q\left\|x^k - x^*\right\|, q \in (0,1)$
  - Сверхлинейная $\left\|x^{k+1} - x^*\right\| \le q_k\left\|x^k - x^*\right\|, q_k \to 0$
  - Квадратичная $\left\|x^{k+1} - x^*\right\| \le q\left\|x^k - x^*\right\|^2$
  - и т.д.

Методы спуска

Определение. Вектор $d \in \mathbb{R}^n$ называется направлением убывания функции $f: \mathbb{R}^n \to \mathbb{R}$ в точке $x \in \mathbb{R}^n$, если для малого η:

$$f(x + \eta d) < f(x)$$

$D_f(x)$ – множество (конус) всех направлений убывания в точке (все вектора, чье скалярное произведение с антиградиентом положительно)

$$\forall d \in D_f(x), \langle \nabla f(x), d \rangle \leq 0$$

Методы спуска:

$$x^{k+1} = x^k + \eta_k d^k, d \in D_f\left(x^k\right), k = 0,1, \dots,$$

$$\eta_k: \{f(x^k)\} \text{ убывает}$$

Каждый метод – свой подход к выбору **шага** и выбору **направления** убывания

Критерии остановки: число шагов, малое изменение целевой функции, близость нормы градиента к 0

Градиентный метод

Направление спуска противоположно градиенту: $d^k = -\nabla f(x)$

- Выбираем начальное приближение $x^0 \in \mathbb{R}^n$
- Каждое следующее приближение определяется по правилу:

$$x^{k+1} = x^k - \eta_k \nabla f(x^k), k = 0,1,\ldots$$

Правила выбора шага:

- априорное задание $\{\eta_k\}_{k=0}^{\infty}$ в виде константы или правил пересчета, например:

$$\eta_k = \eta, \eta_k = \frac{\eta}{1+\beta k}$$

- полная релаксация - *метод наискорейшего спуска*

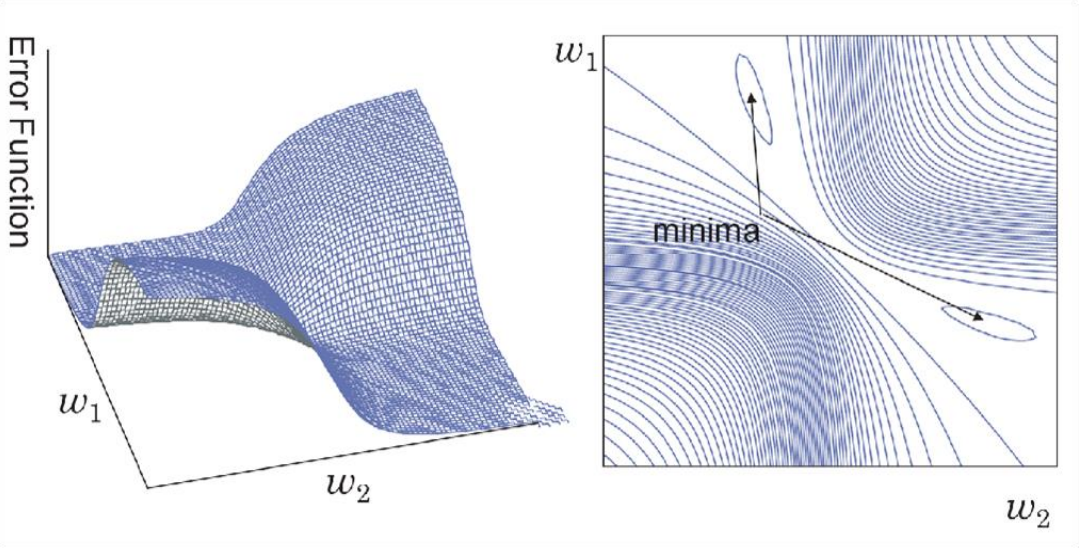
$$\eta_k = \arg\min_{\eta \geq 0} f(x^k - \eta \nabla f(x^k))$$

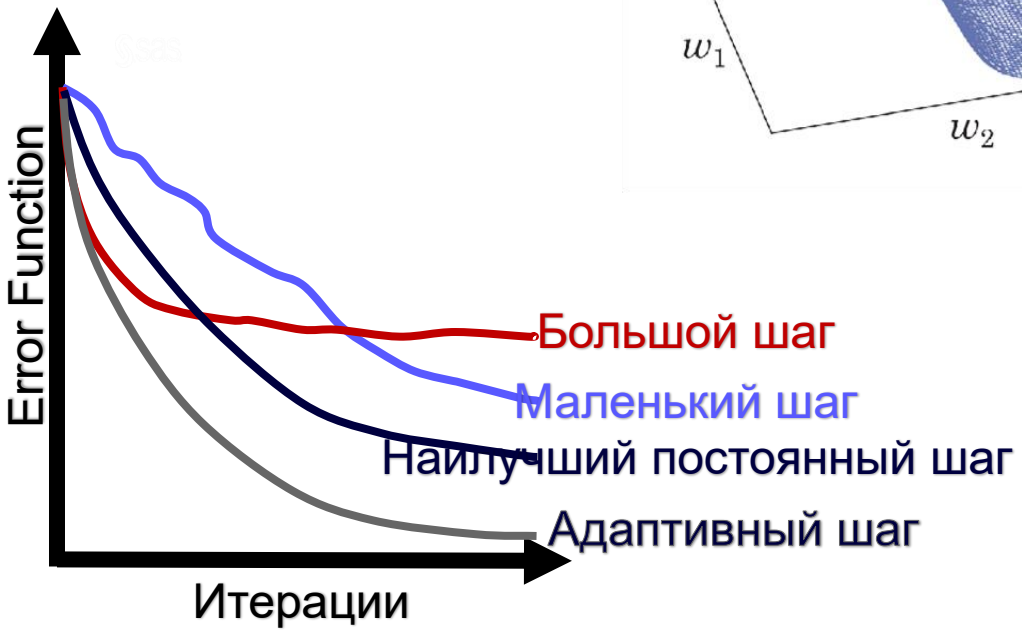
- правило Армихо, Вульфа и другие

Траектория для квадратичной функции

Градиентный метод в машинном обучении

«Поверхность ошибки» и контурная проекция - невыпуклая





Зависимость от выбора шага для невыпуклой функции

Метод Ньютона

Пусть $f(x)$ – выпуклая дважды дифференцируемая функция

По определению дважды дифференцируемой функции (разложим в ряд Тейлора):

$$f(x) - f(x^k) = \langle \nabla f(x^k), x - x^k \rangle + \frac{1}{2}\langle \nabla^2 f(x^k)(x - x^k), x - x^k \rangle + o(\|x - x^k\|^2)$$

Минимизируем квадратичную часть:

$$\langle \nabla f(x^k), x - x^k \rangle + \frac{1}{2}\langle \nabla^2 f(x^k)(x - x^k), x - x^k \rangle \to \min$$

Она выпукла, так как ее гессиан $\nabla^2 f(x^k) \geq 0$

Необходимое и достаточное условие минимума:

$$\nabla f(x^k) + \nabla^2 f(x^k)(x - x^k) = 0$$

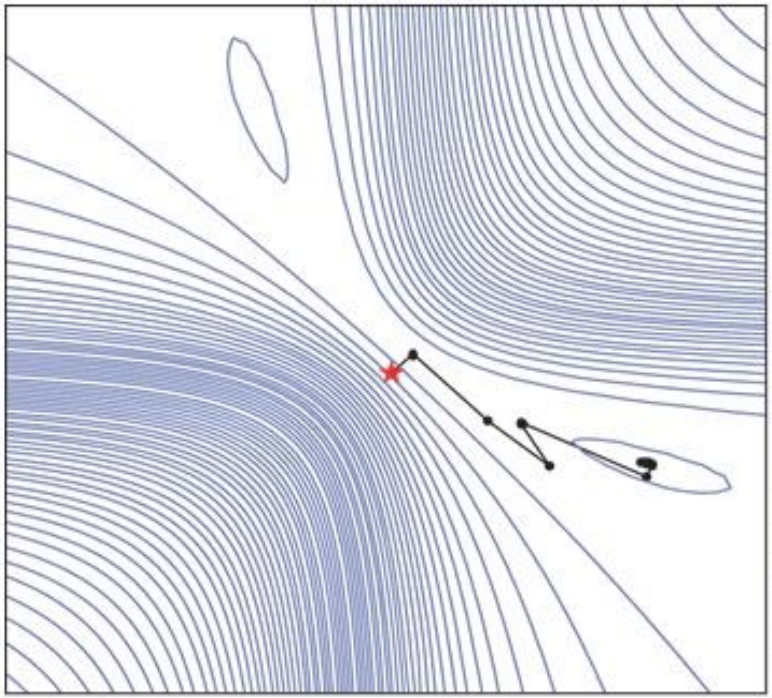
Отсюда получаем метод Ньютона:

$$x^{k+1} = x^k + h^k, h^k = -(\boldsymbol{\nabla^2 f(x^k)})^{-1}\nabla f(x^k)$$

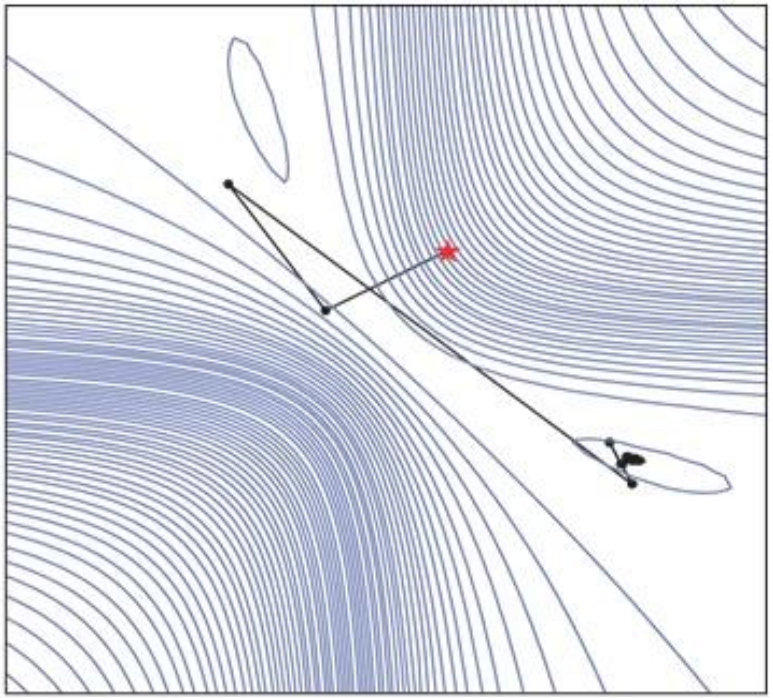
Особенности метода Ньютона

- Нет проблемы **выбора шага**
- **Сходимость:**
  - Если целевая функция дважды дифференцируема, сильно выпуклая, тогда получаемая в методе Ньютона последовательность приближений сходится к точке минимума с квадратичной скоростью.
  - Для невыпуклых функций сходимость гарантирована только при условии близости начальной точки к точке минимума.
- Необходимость выбора **начального приближения**:
  - в окрестности решения, условие близости трудно проверить
- **Вычислительная** трудоемкость, а иногда и нестабильность:
  - нужно вычислять на каждом шаге матрицу вторых производных и затем обратную к ней.
- Желание сохранить быструю скорость сходимости и устранить недостатки привели к разработке множества **модификаций** метода Ньютона и **квазиньютоновских** методов

Пример



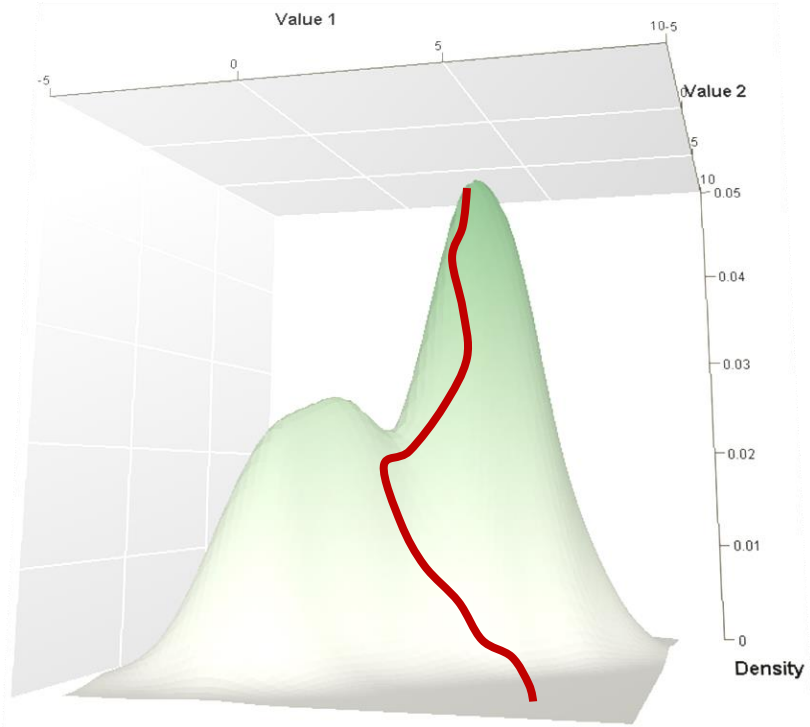
38 итераций



57 итераций

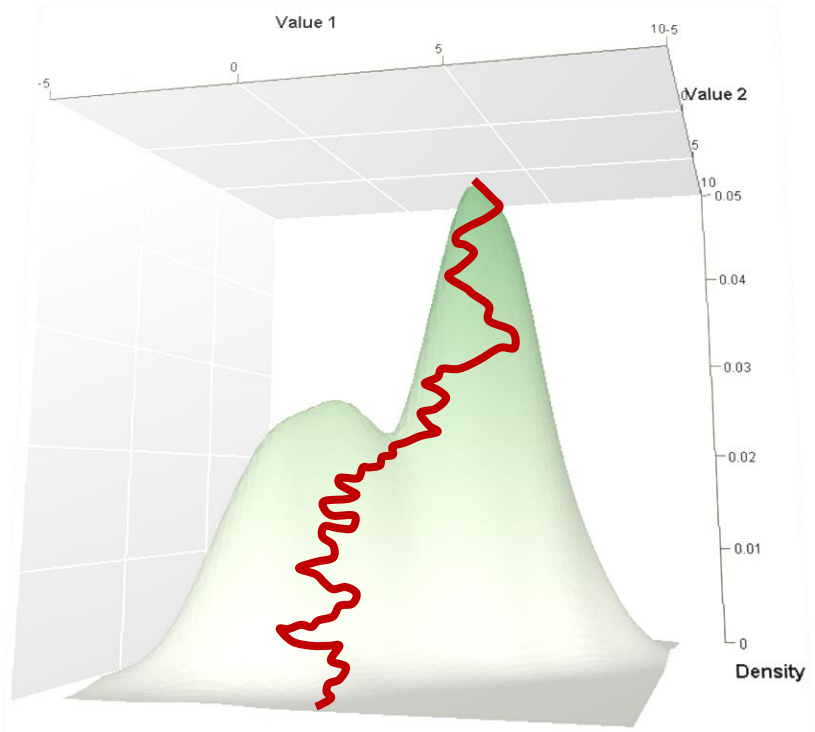
Не нужно задавать длину шага, мало итераций, но много вычислений на каждой – нужно считать матрицу вторых производных и обращать ее

Классические методы оптимизации для задачи машинного обучения



- Целевая функция – эмпирический риск, усредненная сумма значений функций риска для всех наблюдений
- Используют все наблюдения выборки для расчета эмпирического риска, его градиента или приближения гессиана
- Результат – гладкая траектория оптимизации
- Но долго по времени и много по памяти (вся выборка)

Стохастические и пакетные методы обучения



- Аппроксимируют вычисление градиента эмпирического риска (или приближение гессиана) только по части выборки
- Стохастические методы используют только одно наблюдение
- Пакетные – часть наблюдений
- Результат – «хаотическая» траектория (чем больше пакет тем меньше «хаоса»)
- Зато быстро считать и не нужно все брать из памяти – MPP!!!

Важные модификации градиентного метода

- Учет инерции («метод моментов», или импульса) – «сгладить» траекторию за счет предыдущих направлений (импульса), α задает «важность» старого направления

$$w^{k+1} = w^k + v^k, v^k = -(1-\alpha)\eta_k \nabla Q\left(w^k\right) + \boldsymbol{\alpha v^{k-1}}$$

- Метод Нестерова – «сглаживать» после применения старого шага:

$$v^k = -(1-\alpha)\eta_k \nabla Q\left(w^k + \boldsymbol{\alpha v^{k-1}}\right)$$

- Регуляризация (штраф за сложность):

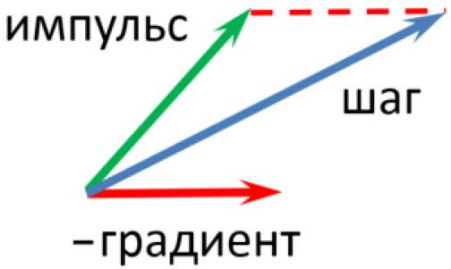
$$\min Q\,(w) + \boldsymbol{\gamma R(w)}$$

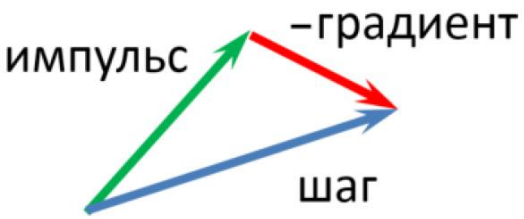
$R(.)$ – гладкая сильно выпуклая функция,

например, регуляризация L_p

γ-константа регуляризации

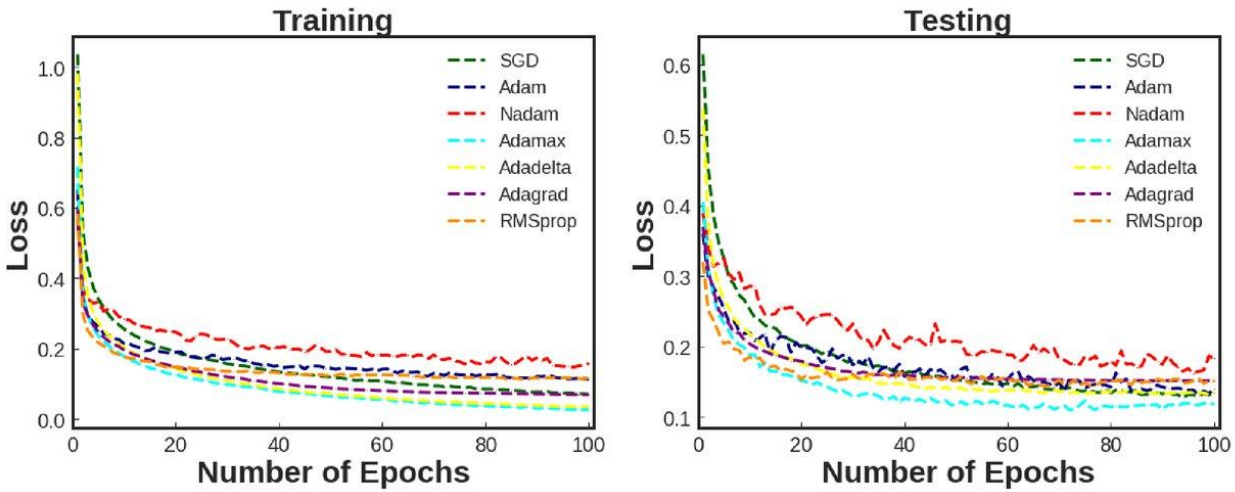
$$v^k = -\eta_k[\nabla Q\left(w^k\right) + \gamma \nabla R\left(w^k\right)]$$





Другие популярные модификации SGD

- RProp – устойчивый к угасанию градиента (в многослойных нейросетях), учитывает знак градиента, но не учитывает его величину (шаг постоянный)
- Adam (Adaptive Moment Estimation) – экспоненциальное сглаживание градиентов
- AdaGrad и RMSProp – индивидуальные скорости обучения для каждого параметра «по ситуации»
- …



Идея градиентного бустинга

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

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

  - 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} :

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

  - Итерационно (**градиентным методом с шагом** α_m) минимизируем Q по $F_m = F_{m-1} - \alpha_m \nabla Q_m$, 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 $F_m = F_{m-1} + \alpha_m b_m$ => 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 $-\boldsymbol{\nabla Q_m}$ **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):

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

Градиентный бустинг деревьев

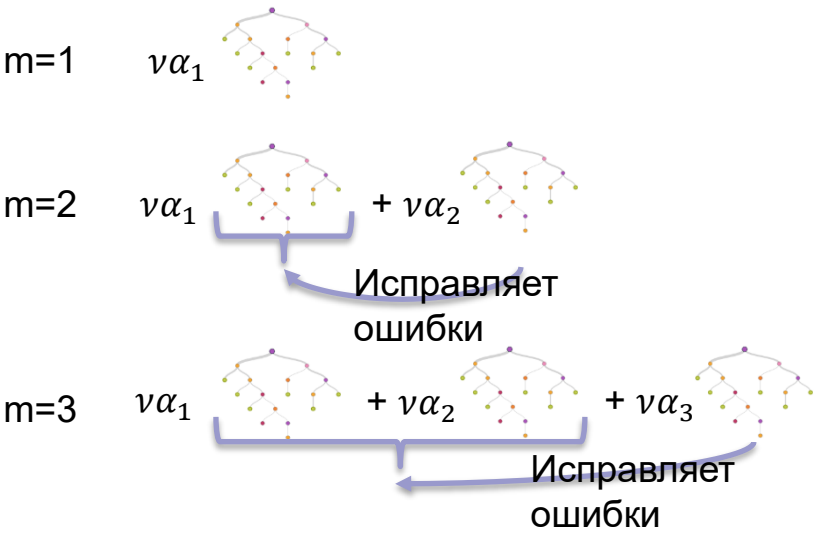
- Финальная модель (M – число итераций):

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

- Каждая базовая модель – дерево:

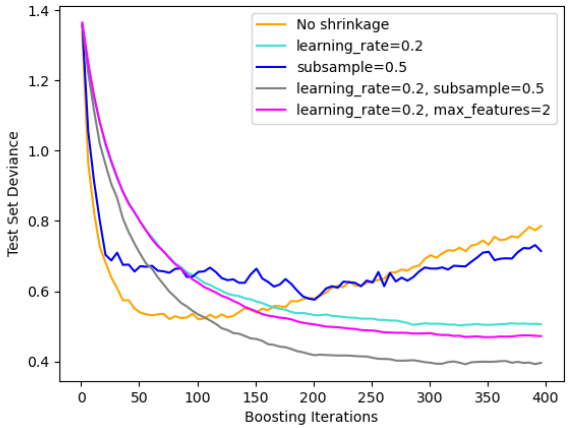
$$T_m(x) = \sum_{R \in R_m} \gamma_R \, I[x \in R]$$

  - где R_m - множество регионов, соответствующих листьям дерева, а γ_R - прогноз отклика в листе
  - Каждая следующая модель $T_m(x)$ обучается на «псевдоостатках» 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



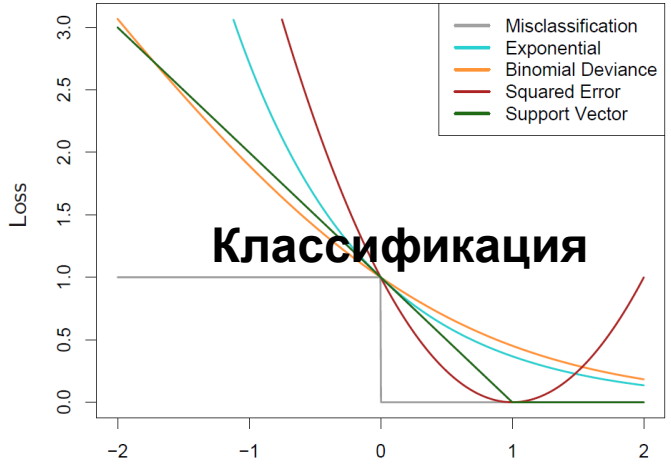
Борьба с переобучением

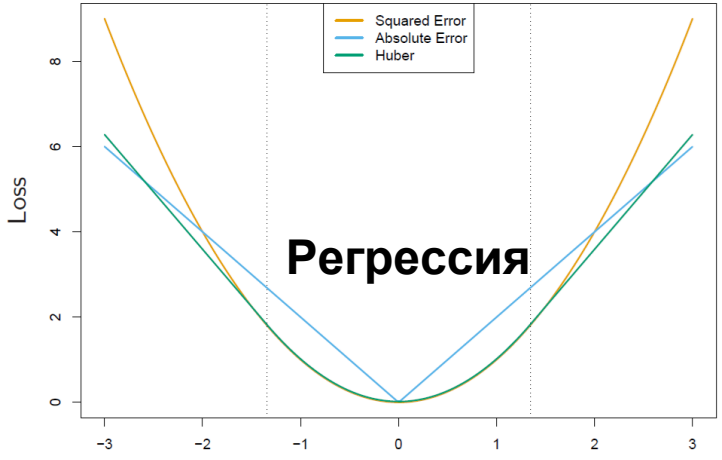
- Градиентный бустинг склонен к переобучению в условиях шума
- Методы борьбы с переобучением:
  - **Стохастический градиентный бустинг** - случайные подвыборки (можно использовать OOB оценки) меньшего размера для обучения базовых моделей на каждой итерации (иногда используют бутсрепинг): $Z_m = (x_i, -\nabla Q_m(x_i))_{i=1}^{n<l}$
  - **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 $\text{sign}(y_i - a(x_i))$
K классов	K 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 $k = \text{argmax}_s[p_s(x)]$
  - $g_k(x)$ - дискриминантная функция класса k
  - $p_k(x) = \text{softmax}\big(g_1(x), \dots, g_K(x)\big) = \frac{e^{g_k(x)}}{\sum_{s=1}^{K} e^{g_s(x)}}$
  - Кросс энтропия: $L\big(y, p(x)\big) = -\sum_{k=1}^{K} I[y = k] \log\big(p_k(x)\big) = -\sum_{k=1}^{K} I[y = k]\, g_k(x) + \log\big(\sum_{s=1}^{K} e^{g_s(x)}\big)$
  - Псевдоостаток для класса k: $\boldsymbol{-L'\big(y, p_k(x)\big) = I[y_i = C_k] - p_k(x_i)}$
- Бинарный случай:
  - $Y = \{0,1\}, p(x) = p_1(x) = P(Y = 1|x) = \frac{1}{1+e^{-g(x)}}$, $p_0(x) = 1 - p(x)$
  - $L\big(y, p(x)\big) = y\log\big(p(x)\big) + (1 - y)\log\big(1 - p(x)\big)$, и если $g(x) \equiv a(x) \Rightarrow$
    $L\big(y, a(x)\big) = \log(1 + e^{-a(x)y}) \Rightarrow$
    Псевдоостаток $\boldsymbol{L'\big(y, a(x)\big) = -y/(1 + e^{-a(x)y})}$

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
```

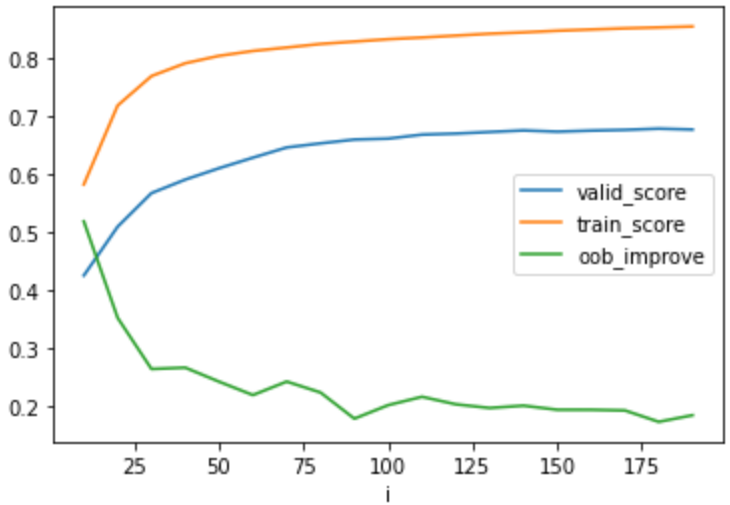
```
((20640, 8), (20640,), ['MedHouseVal'])
```

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

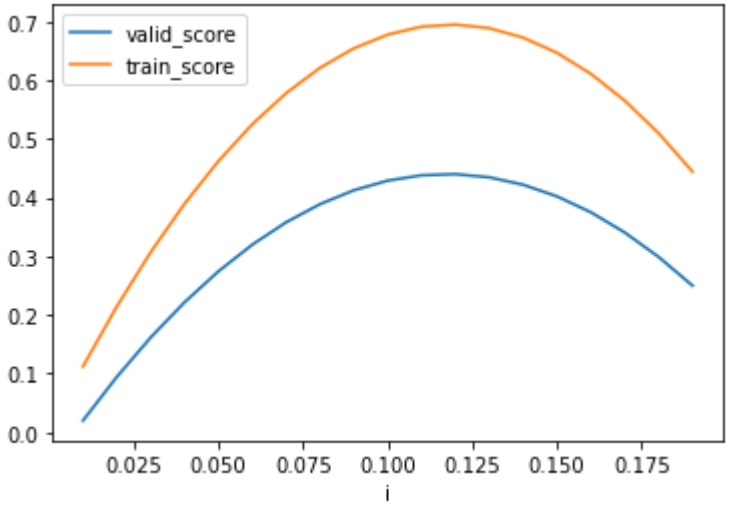
```
n_estimators = 100
boosting = GradientBoostingRegressor(n_estimators=n_estimators, learning_rate=0.1,
                                     max_depth=5,
                                     max_leaf_nodes=10,
                                     subsample=0.75, # stohastic if <1.0
                                     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)

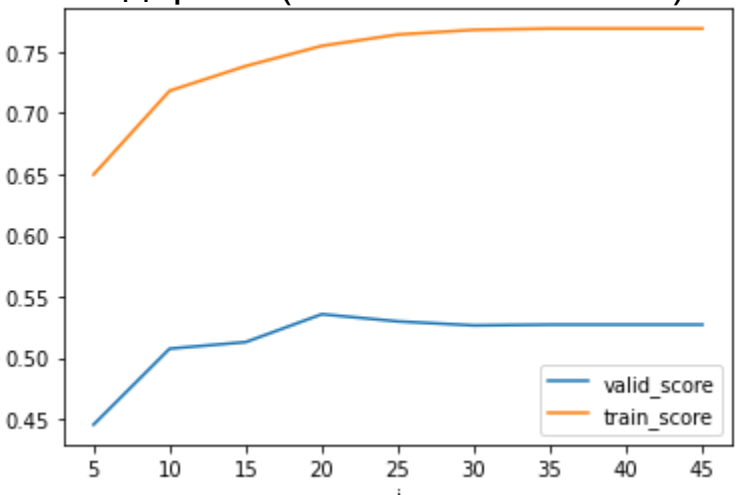
Качество от размера ансамбля



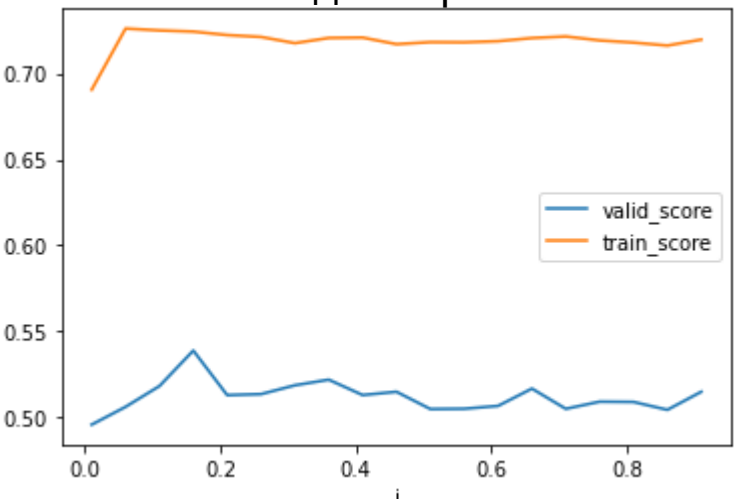
Качество от shrinkage



Качество от сложности дерева (max число листьев)



Качество от размера подвыборки

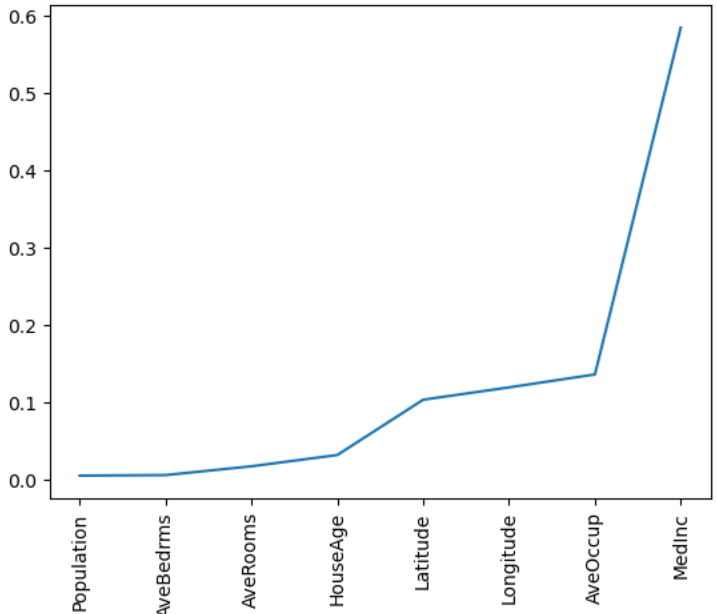


Градиентный бустинг (пример)

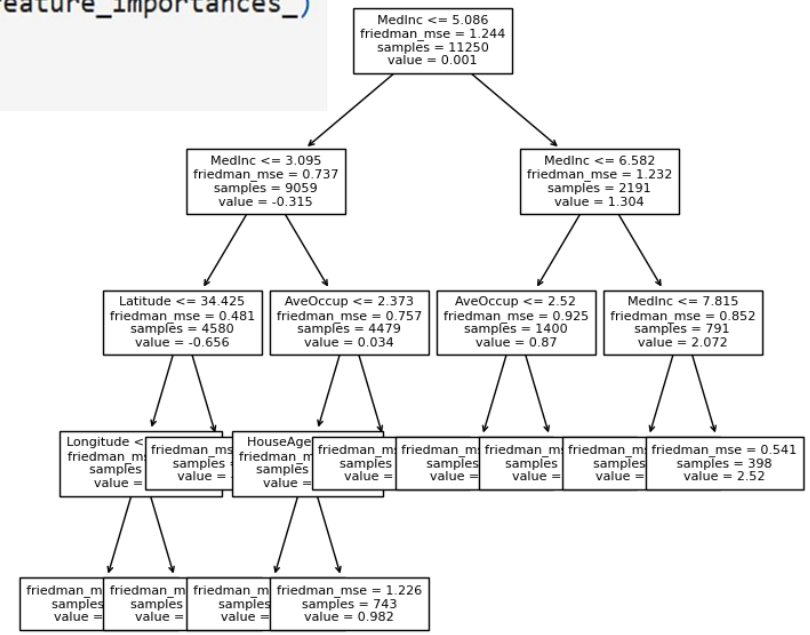
- Оценить важность переменных по ансамблю:

```
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

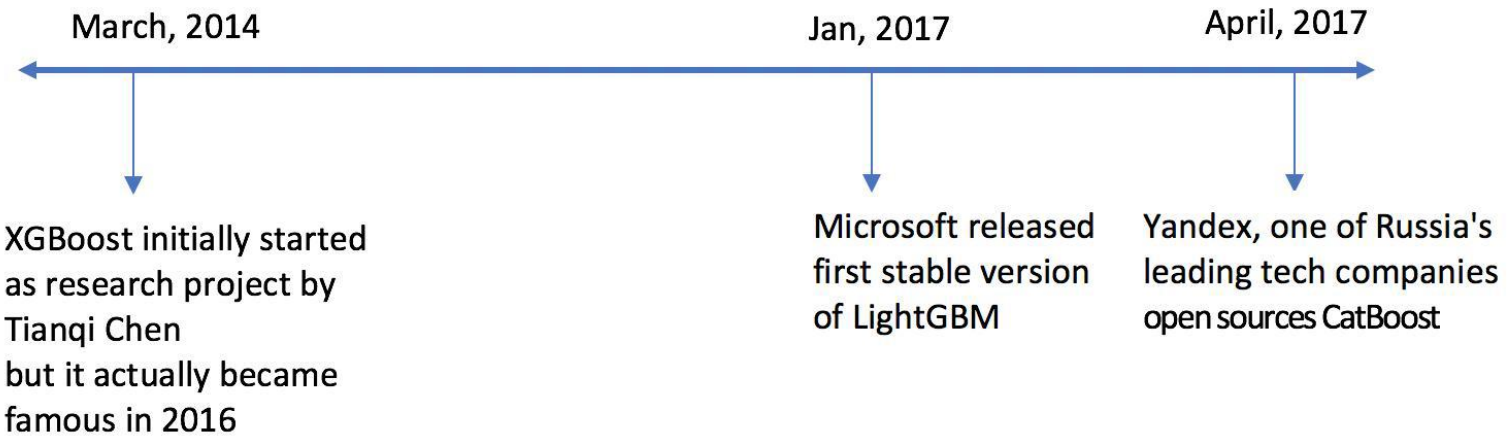


- Можно «добраться» до каждого дерева в ансамбле




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

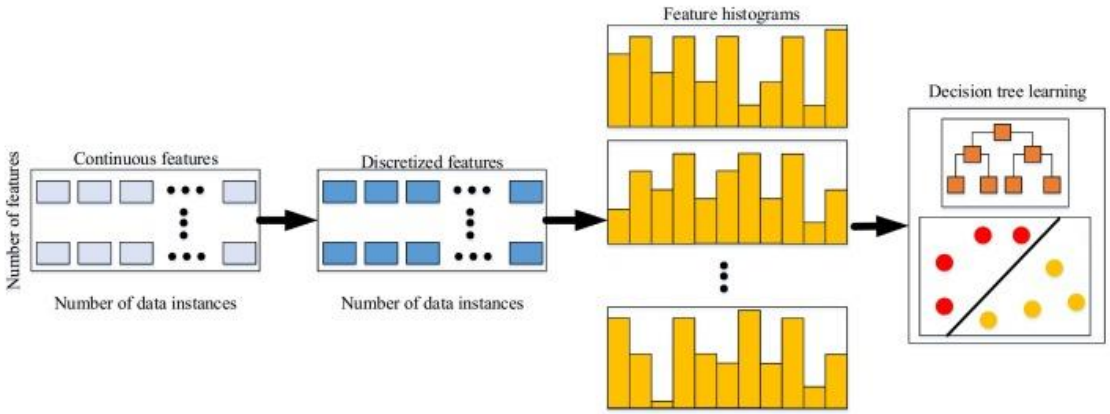
Современные алгоритмы бустинга



- Общие особенности:
  - Распараллеливание, поддержка GPU, TPU, возможности AutoML
  - Возможность дообучения для больших объемов данных
  - Разные стандартные и пользовательские метрики и функции потерь
  - Бэгинг, 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 бэгинга (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

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

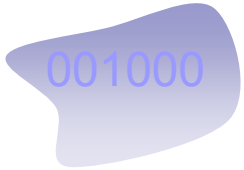
  - Для числового отклика $\rho = E(y_i | (x_i, y_i) \in Z_m)$, **c** – параметр

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

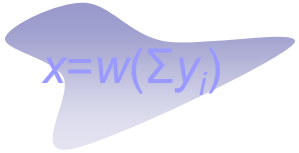
- Жадная **группировка значений категориальных** переменных:
  - Кодируем по порядку часто встречающиеся комбинации значений признаков, а не их декартово произведение
  - Эффективно при One-hot-encoding

Подходы к кодировке категориальных признаков

Случайное кодирование



Бинарное кодирование



Преобразование
с учётом отклика

Unsupervised:

- Backward Difference Contrast [2][3]
- BaseN [6]
- Binary [5]
- Gray [14]
- Count [10]
- Hashing [1]
- Helmert Contrast [2][3]
- Ordinal [2][3]
- One-Hot [2][3]
- Rank Hot [15]
- Polynomial Contrast [2][3]
- Sum Contrast [2][3]

Supervised:

- CatBoost [11]
- Generalized Linear Mixed Model [12]
- James-Stein Estimator [9]
- LeaveOneOut [4]
- M-estimator [7]
- Target Encoding [7]
- Weight of Evidence [8]
- Quantile Encoder [13]
- Summary Encoder [13]

https://github.com/scikit-learn-contrib/category_encoders

Преобразование с учётом отклика

Level	N_i	ΣY_i	p_i
A	**1562**	**430**	**0.28**
B	**970**	**432**	**0.45**
C	**223**	**45**	**0.20**
D	**111**	**36**	**0.32**
E	**85**	**23**	**0.27**
F	**50**	**20**	**0.40**
G	**23**	**8**	**0.35**
H	**17**	**5**	**0.29**
I	**12**	**6**	**0.50**
J	**5**	**5**	**1.00**

«редкие значение» переменной - источник нестабильности, недостоверности в модели

Level* –** различные значения переменной ***X

N_i**–** число наблюдений, что ***X*** принимает ***i***-е значение

Σy_i - сумма бинарных откликов для наблюдений, где ***X*** принимает ***i***-е значение

$p_i = \Sigma y_i / N_i$ - условная вероятность положительного отклика, если ***X*** принимает ***i***-е значение

Преобразование с учётом отклика

Level	N_i	ΣY_i	p_i
J	5	5	1.00
I	12	6	0.50
B	970	432	0.45
F	50	20	0.40
G	23	8	0.35
D	111	36	0.32
H	17	5	0.29
A	1562	430	0.28
E	85	23	0.27
C	223	45	0.20

Сортируем по $\boldsymbol{p_i}$, если значения ***X*** «рядом» после сортировки, то можно предположить, что они «похоже» влияют на отклик

Кодирование категориального признака порядковой (ординальной) переменной

X'	N_i	ΣY_i	p_i
1	5	5	1.00
2	12	6	0.50
3	970	432	0.45
4	50	20	0.40
5	23	8	0.35
6	111	36	0.32
7	17	5	0.29
8	1562	430	0.28
9	85	23	0.27
10	223	45	0.20

Можем отобразить категориальный ***X*** на порядковую шкалу (новая переменная ***X'***), но не учитываем насколько похожи отклики и не решается проблема редких значений

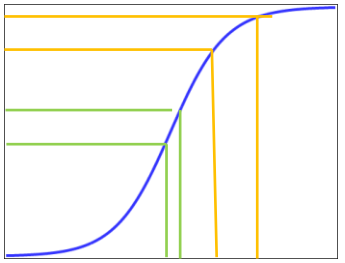
Шансы

Level	N_i	ΣY_i	p_i	$\log(p_i/1-p_i)$
J	5	5	1.00	.
I	12	6	0.50	0.00
B	970	432	0.45	-0.10
F	50	20	0.40	-0.18
G	23	8	0.35	-0.27
D	111	36	0.32	-0.32
H	17	5	0.29	-0.38
A	1562	430	0.28	-0.42
E	85	23	0.27	-0.43
C	223	45	0.20	-0.60

$\Delta p_i = 0.05 \Rightarrow \Delta\text{logit}(p_i)=0.1$

$\Delta p_i = 0.05 \Rightarrow \Delta\text{logit}(p_i)=0.11$

Рассмотрим логарифм шансов положительного отклика для ***i***-го значения ***X***, т.е. **logit** от p_i, сортировка не меняется, но более корректно учитываются различия в областях определенности (около 0 и 1) и неопределенности



Группировка значений переменной по шансам

Level	N_i	ΣY_i	p_i	$\log(p_i/1-p_i)$
J	5	5	1.00	.
I	12	6	0.50	0.00
B	970	432	0.45	-0.10
F	50	20	0.40	-0.18
G	23	8	0.35	-0.27
D	111	36	0.32	-0.32
H	17	5	0.29	-0.38
A	1562	430	0.28	-0.42
E	85	23	0.27	-0.43
C	223	45	0.20	-0.60

Можем агрегировать (например, с помощью одномерной кластеризации) «похожие» значения ***X***, объединив их в однородные (с точки зрения поведения отклика) группы …

Группировка значений переменной по шансам

X”	N_i	ΣY_i	p_i	$\log(p_i/1-p_i)$
CL1	1037	463	0.45	-0.09
CL2	134	44	0.33	-0.31
CL3	1664	458	0.28	-0.42
CL4	223	45	0.20	-0.60

… и создать новую категориальную переменную ***X”,*** без редких уровней и с меньшим числом различных значений – уменьшаем число степеней свободы модели, увеличиваем стабильность модели и уменьшаем шанс переобучиться

Weight of evidence и шансы

- **Формула Байеса**:
  - Обозначения : маленькие p – плотности, большие P – вероятности
  - $p(x)$ - плотность распределения наблюдений в пространстве признаков
  - $P(y)$ - априорная и $P(y|x)$ - апостериорная вероятности классов
  - $p(x|y)$ – функция правдоподобия
  - Совместная плотность распределения:

$$\boldsymbol{p(x,y) = p(x)P(y|x) = P(y)p(x|y) \Rightarrow P(y|x) = P(y)p(x|y)/p(x)}$$

- Логарифм условных шансов:

$$\log\left(\frac{P(y=1|x)}{P(y=0|x)}\right) = \log\left(\frac{P(y=1)p(x|y=1)/p(x)}{P(y=0)p(x|y=0)/p(x)}\right) = \log\left(\frac{P(y=1)}{P(y=0)}\right) + \log\left(\frac{p(x|y=1)}{p(x|y=0)}\right)$$

- WOE и «наивное» предположение (независимость переменных):

$$\log\left(\frac{P(y=1|x)}{P(y=0|x)}\right) = \log\left(\frac{P(y=1)}{P(y=0)}\right) + \sum_{j=1}^{p} \mathrm{WOE}(x_j), \quad \text{где } \mathrm{WOE}(x_j) = \log\left(\frac{p(x_j|y=1)}{p(x_j|y=0)}\right)$$

Априорные шансы (log равен 0, если выборка «сбалансирована»)

Вклад шансов каждой из p переменных

Weight of Evidence для категориальной переменной

- Пусть в задаче с бинарным откликом категориальная (или дискретизированная) переменная x_j принимает k различных значений $\{v_1, \dots, v_k\}$, тогда (опять по формуле Байеса):
  - $WOE(x_j)$ - j–й переменной: $WOE(x_j) = \sum_{i=1}^{k} \mathrm{WOE}_i(x_j)$,
  - где $WOE_i(x_j)$ - weight of evidence i–го значения v_i

$$\mathrm{WOE}_i(x_j) = \log\left(\frac{P(x_j = v_i | y = 1)}{P(x_j = v_i | y = 0)}\right) = \log\left(\frac{count((y = 1) \text{ and } (x_j = v_i))/count(y = 1)}{count((y = 0) \text{ and } (x_j = v_i))/count(y = 0)}\right)$$

- Интерпретация WOE:
  - ***WOE**i* **> 0***: (ОШ>1) i-e значение связано с более высоким шансом положительного отклика
  - ***WOE**i* **< 0***: (ОШ<1) i-e значение связано с более низким шансом положительного отклика
  - ***WOE**i* **= 0***: (ОШ=1) i-e значение не влияет на уровень отклика
  - ***WOE*** в целом оказывает *предиктивную силу* категориальной (или дискретезированной) переменной и ее отдельных значений

Weight of Evidence

Level	N_i	ΣY_i	p_i	WOE_i
J	5	5	1.00	.
I	12	6	0.50	0.71
B	970	432	0.45	0.49
F	50	20	0.40	0.3
G	23	8	0.35	0.08
D	111	36	0.32	-0.03
H	17	5	0.29	-0.17
A	1562	430	0.28	-0.26
E	85	23	0.27	-0.28
C	223	45	0.20	-0.67
	3058	1010		0.17

Не решает проблему редких уровней

Информационная важность переменных

- Дивергенция Кульбака-Лейблера (различающая информация, относительная энтропия и другие термины):
  - Ассиметричная мера расхождения двух распределений
  - Для дискретных распределений P и Q: $D_{KL}(P||Q) = \sum_i \ p_i \log(p_i/q_i)$
  - Симметричный вариант: $D_{KL}(P;Q) = D_{KL}(P||Q) + D_{KL}(Q||P)$
- IV **(информационная важность** или информационный индекс) :
  - Симметричная дивергенция (расстояние) Кульбака-Лейблера, которое показывает насколько отличаются распределения переменной (отдельных значений переменной) внутри положительного и отрицательного классов:
  - Пусть в задаче с бинарным откликом категориальная переменная x принимает k различных значений $\{v_1, \dots, v_k\}$, тогда:

$$IV(x) = \sum_{j=1}^{k} (P(x = v_j | y = 1) - P(x = v_j | y = 0)) WOE_j(x)$$

Information Value

Level	N_i	ΣY_i	p_i	IV_i
J	5	5	1.00	.
I	12	6	0.50	0.0021
B	970	432	0.45	0.0809
F	50	20	0.40	0.0015
G	23	8	0.35	0
D	111	36	0.32	0
H	17	5	0.29	0.0002
A	1562	430	0.28	0.033
E	85	23	0.27	0.0021
C	223	45	0.20	0.0284
	3058	1010		0.1482

- Эвристические пороги на IV:
  - Меньше 0.02 – незначимая переменная
  - 0.02 – 0.10 низкая прогнозная сила
  - 0.10 – 0.30 средняя прогнозная сила
  - 0.30 – 0.50 высокая прогнозная сила
  - Больше 0.50 – что-то пошло не так

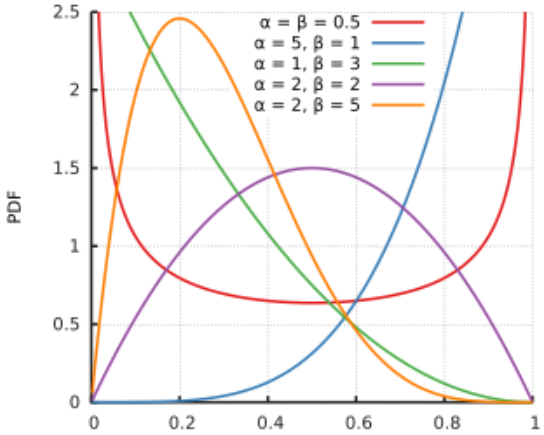
Сглаженное WOE для борьбы с редкими значениями

Level	N_i		ΣY_i		p_i	SWOE
J	5	+24	5	+8	0.45	0.5
I	12	+24	6	+8	0.39	0.25
B	970	+24	432	+8	0.44	0.48
F	50		20		0.38	0.21
G	23	+24	8	+8	0.34	0.05
D	111	+24	36	+8	0.33	-0.02
H	17	+24	5	+8	0.32	-0.06
A	1562		430		0.28	-0.26
E	85	+24	23	+8	0.28	-0.22
C	223	+24	45	+8	0.21	-0.59

- Для «исправления» ситуации с редкими значениями:
  - Добавим для каждого значения переменной «виртуальный» набор наблюдений (фиксированного размера) с вероятностью положительного отклика равной априорной.
  - Чем более редкий уровень, тем выше влияние априорного распределения.

Бета распределение для кодирования категориальных предикторов по выборке

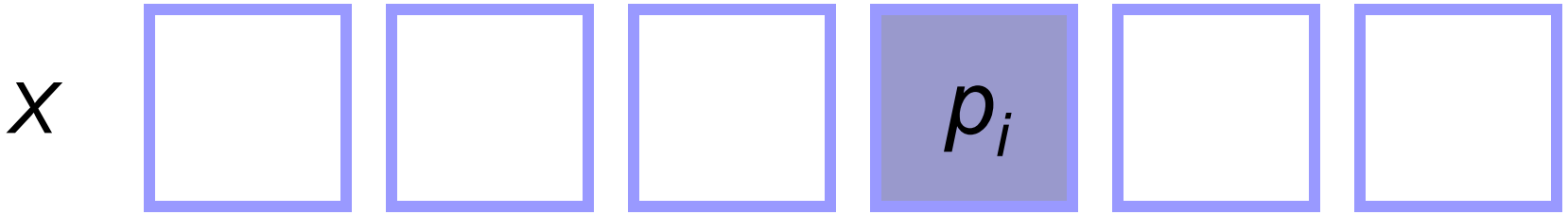
- Бета распределение:
  - двухпараметрическое (альфа и бета)
  - мат. ожидание: $\frac{a}{a+b}$
  - используется для моделирования случайных величин, заданных на интервале.

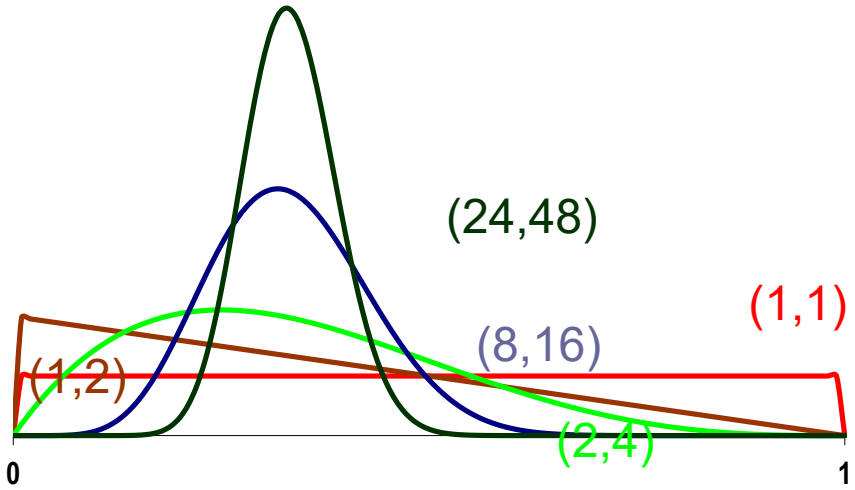


- Моделируем:
  - условную вероятность положительного отклика для фиксированного значения категориальной переменной как случайную величину $p_i \sim \text{Beta}\,(a_s, b_s)$,
  - a_0, b_0 - параметры априорного распределения
  - a_s, b_s пересчитываются последовательно, проходя всю выборку, где категориальная переменная принимает i-e значение, при этом …
  - $a_{s+1} = a_s + 1$, если встретили наблюдение с откликом $y = 1$,
  - $b_{s+1} = b_s + 1$, если встретили наблюдение с откликом $y = 0$,

Сглаженный WOE

Значения X

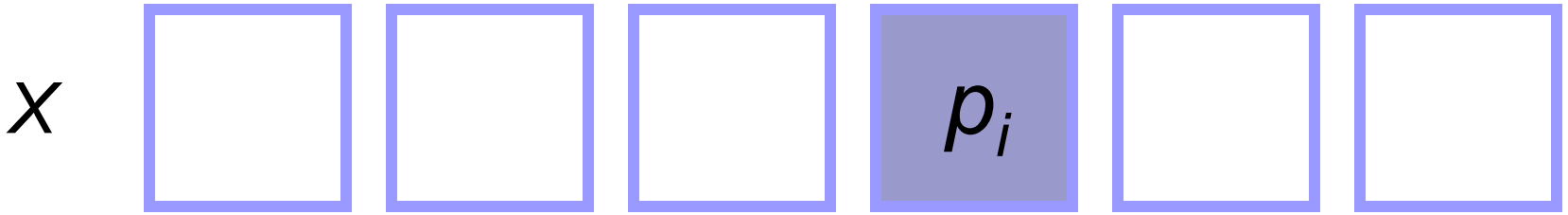


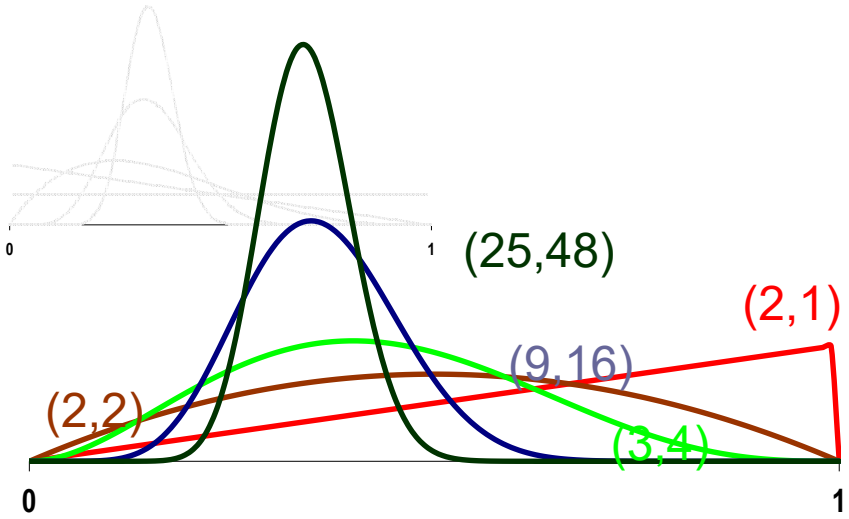


$p_i \sim \text{Beta}\,(a, b)$

Сглаженный WOE

Значение X



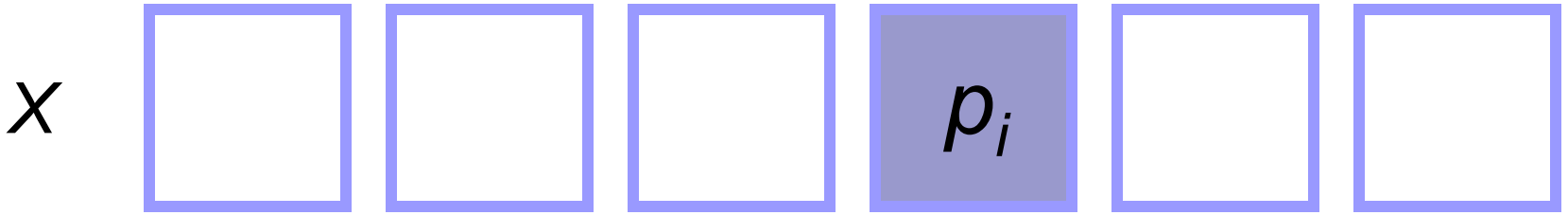


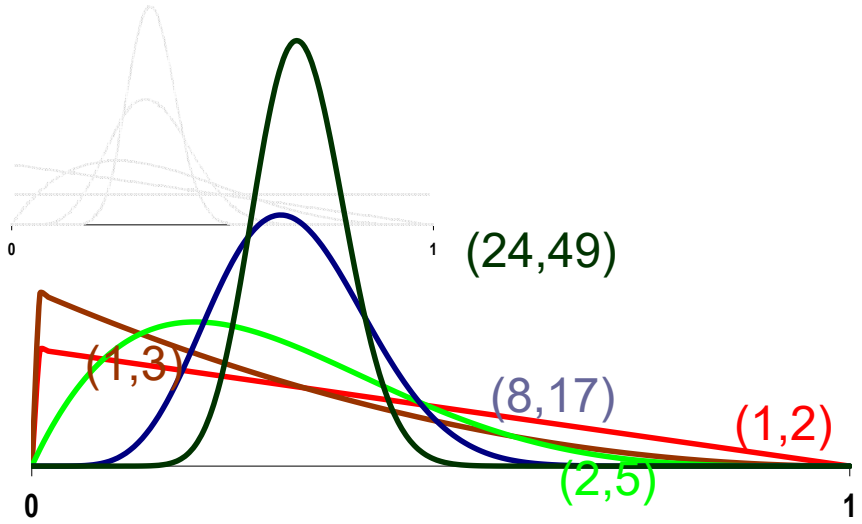
$$p_i \sim \text{Beta}\,(a, b)$$

$$p_i | y = 1 \sim \text{Beta}\,(a + 1, b)$$

Сглаженный WOE

Значения X





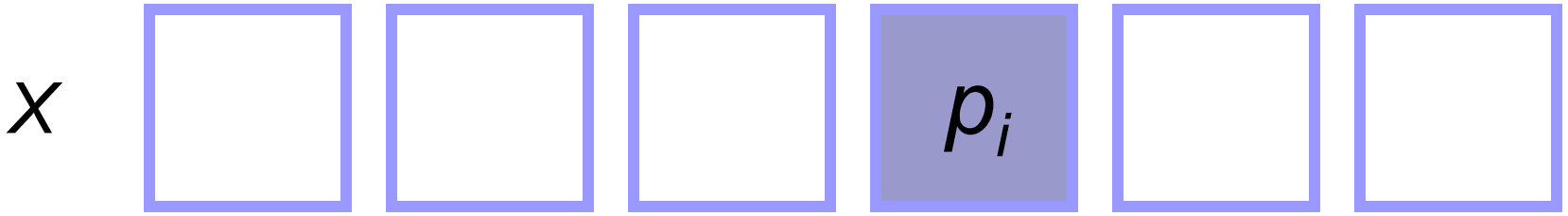
$$p_i \sim \text{Beta}\,(a, b)$$

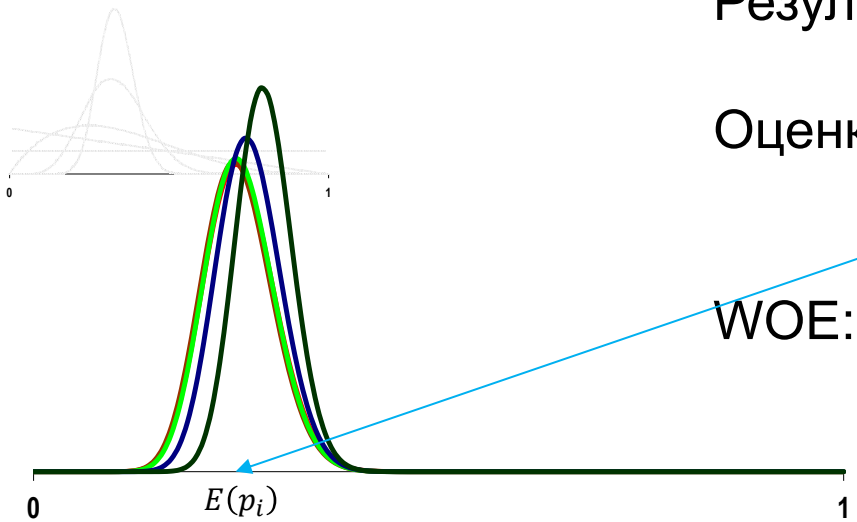
$$p_i | y = 1 \sim \text{Beta}\,(a+1, b)$$

$$p_i | y = 0 \sim \text{Beta}\,(a, b+1)$$

Сглаженный WOE

Значения X





Результат:

$$p_i \sim \text{Beta}\,(a + n_1, b + n_0)$$

Оценка:

$$E(p_i) = \frac{n_1 + a}{n_1 + n_0 + a + b}$$

WOE:

$$\log\left(\frac{E(p_i)}{1 - E(p_i)}\right)$$

Наблюдаемое число событий, если x принимает i-е значение

$$SWOE_i(x) = \log\left(\frac{n_1 + c\rho_1}{n_0 + c(1 - \rho_1)}\right)$$

Априорная вероятность события

Параметр регуляризации

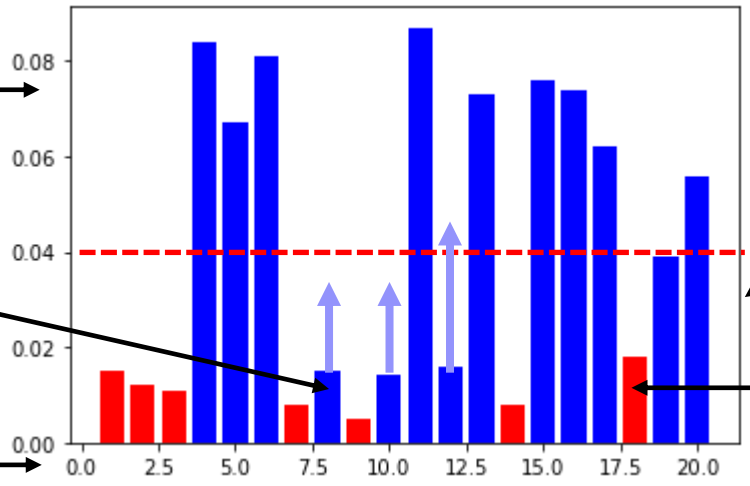
Градиентный sampling

- 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».

Ось y – градиенты
$\nabla Q = \frac{\partial L(y_i, F_{m-1})}{\partial F_{m-1}}(x_i)$

«Усиленные» неважные примеры

Ось x –выборка
$Z = \{x_1, \dots, x_l\}$



Порог «важных» примеров ($\nabla Q \geq 0.04$)

Выброшенные неважные примеры

LightGBM (пример)

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

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

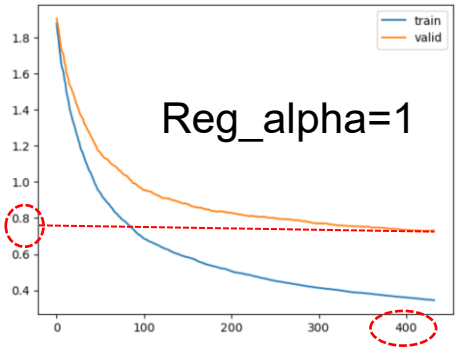
```
lgbm_classifier = LGBMClassifier(boosting_type="gbdt", # dart, rf
                                 num_leaves=10,
                                 max_depth=-1,
                                 learning_rate=0.05,
                                 min_child_samples=20, # min_samples_leaf
        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

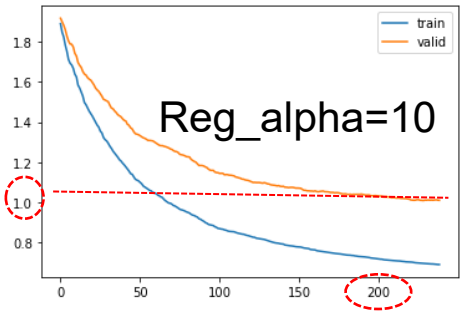
```
history = {}
lgbm_classifier.fit(X_train, y_train, feature_name=covtype.feature_names,
                    eval_set=[(X_test, y_test), (X_train, y_train)],
                    callbacks=[early_stopping(stopping_rounds=10), record_evaluation(history)])
```

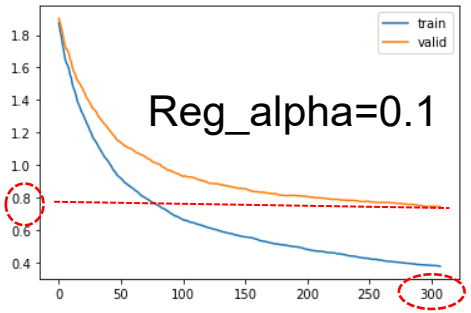
```
Training until validation scores don't improve for 10 rounds
Early stopping, best iteration is:
[423]   training's multi_logloss: 0.349899      valid_0's multi_logloss: 0.728702
```

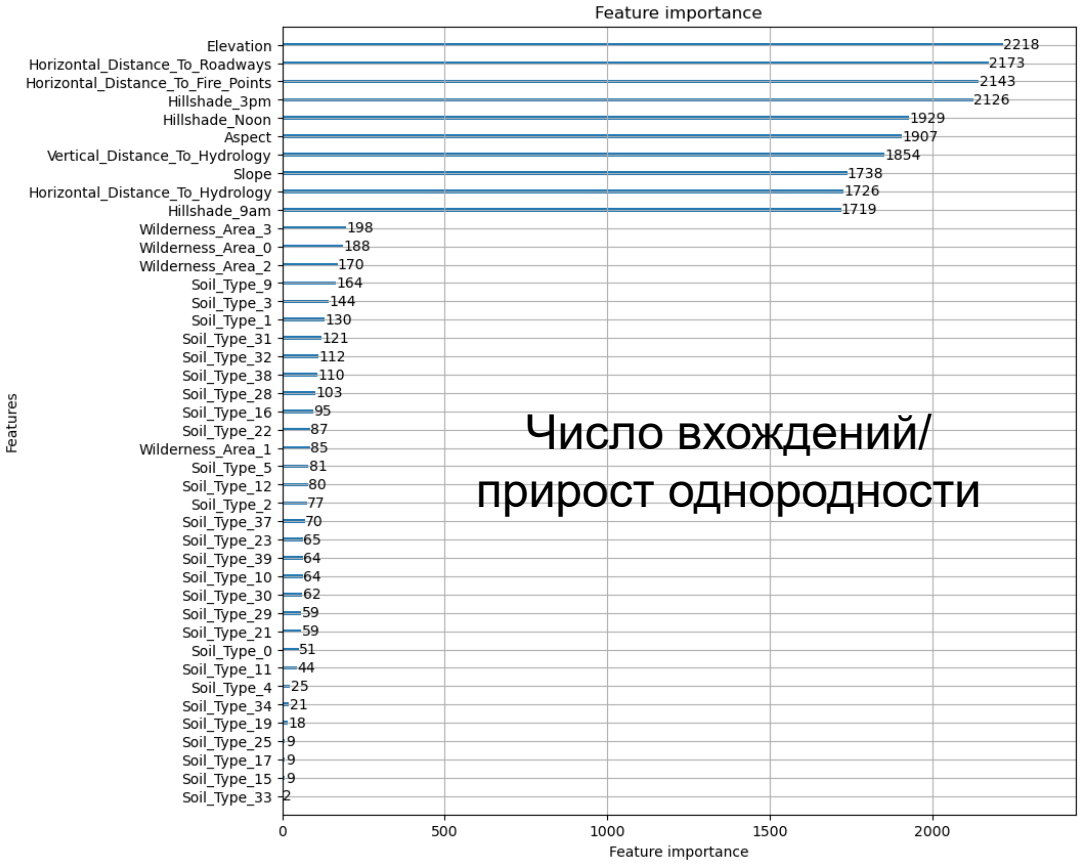
LightGBM (пример)

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





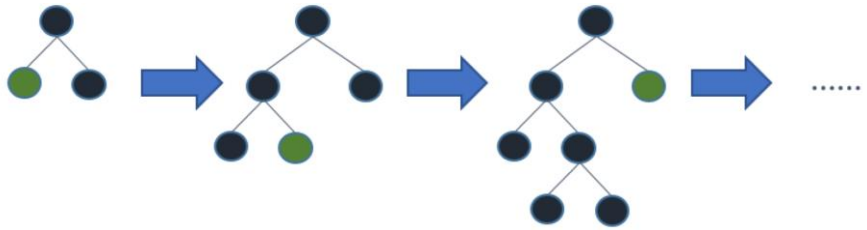




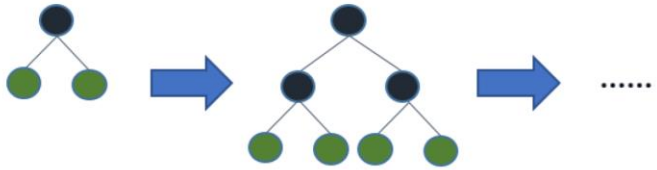

```
plot_importance(lgbm_classifier, figsize=(10, 10));
```

Упрощения роста дерева

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



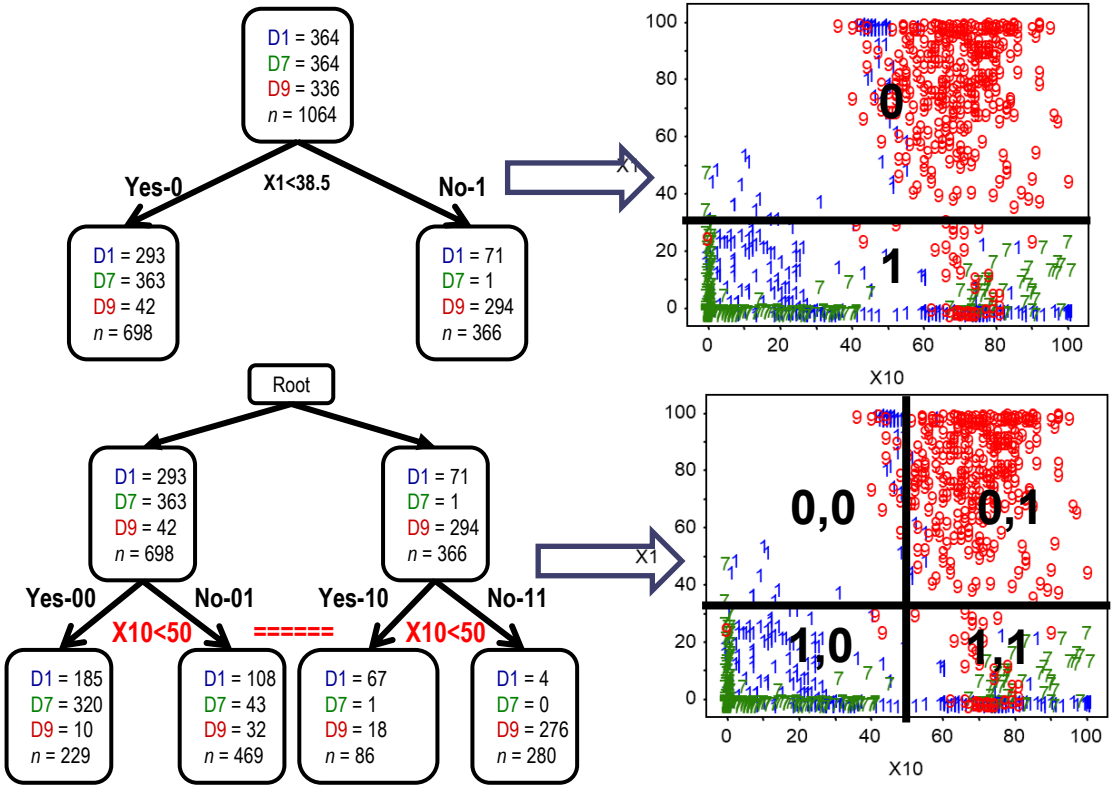
  - **«в ширину»** (level-wise) – упрощенный вариант, на каждом шаге плюс уровень для всех текущих листьев



  - **«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

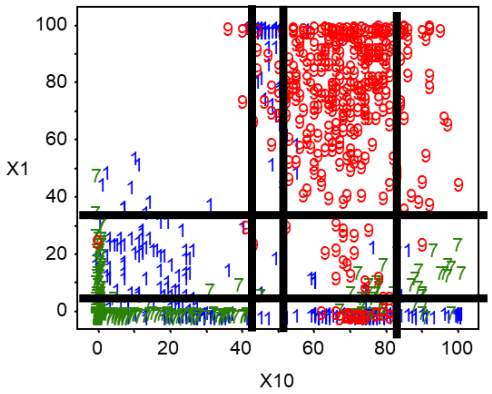


- Решающая таблица:
  - Для дерева глубины D пространство X делится на 2^D ячеек с решающим правилом:

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

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

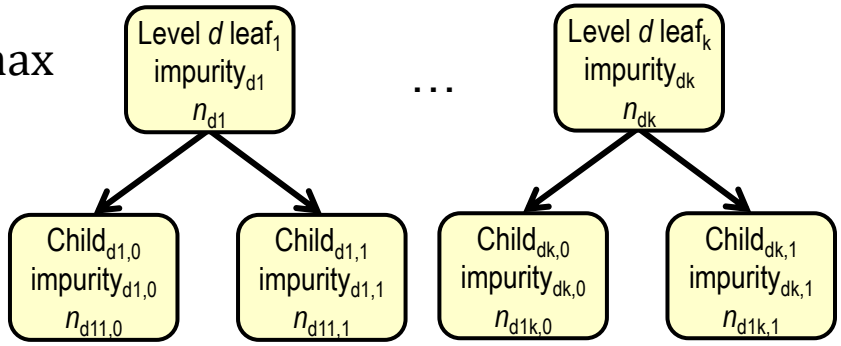
$b_1(x), \dots, b_D(x)$ – вектор прогнозов определяет «координаты» ячейки



Алгоритм обучения (бинарного) ODT

- **Алгоритм разбиения** по уровням (для уровня d)
  - Сформировать множество *гипотез* $\{f_i\}$ *для разбиения* по всем признакам, $f_i: X \to \{0,1\}$ 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):

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



$leaves_d$ - 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

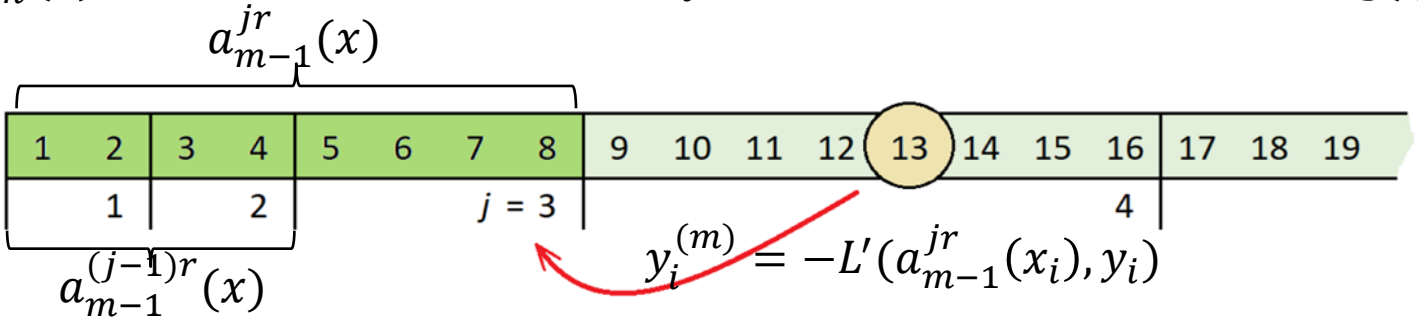
  - Дорастить дерево «в ширину»: каждый лист уровня d превращается во внутренний узел с двумя новыми ветвями

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 $\boldsymbol{a_{m-1}^{(i)}}$**, который не учился на** $\boldsymbol{x_i}$
  - Строить обучающие подвыборки **последовательно** увеличивая размер (например, удваивая длину, тогда нужно не $O(l)$ моделей, а $O\big(\log(l)\big)$)
  - 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** $\boldsymbol{\alpha_m^*}$, 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**
  $G \leftarrow (grad_r(i)$ for $i = 1..n)$;
if $Mode = Ordered$ **then**
  $G \leftarrow (grad_{r,\sigma_r(i)-1}(i)$ for $i = 1..n)$;
$T \leftarrow$ empty tree;
foreach *step of top-down procedure* **do**
  foreach *candidate split* c **do**
    $T_c \leftarrow$ add split c to T;
    if $Mode = Plain$ **then**
      $\Delta(i) \leftarrow \text{avg}(grad_r(p)$ for $p : leaf_r(p) = leaf_r(i))$ for $i = 1..n$;
    if $Mode = Ordered$ **then**
      $\Delta(i) \leftarrow \text{avg}(grad_{r,\sigma_r(i)-1}(p)$ for $p : leaf_r(p) = leaf_r(i), \sigma_r(p) < \sigma_r(i))$ for $i = 1..n$;
    $loss(T_c) \leftarrow \cos(\Delta, G)$
  $T \leftarrow \arg\min_{T_c}(loss(T_c))$
if $Mode = Plain$ **then**
  $M_{r'}(i) \leftarrow M_{r'}(i) - \alpha\,\text{avg}(grad_{r'}(p)$ for $p : leaf_{r'}(p) = leaf_{r'}(i))$ for $r' = 1..s, i = 1..n$;
if $Mode = Ordered$ **then**
  $M_{r',j}(i) \leftarrow M_{r',j}(i) - \alpha\,\text{avg}(grad_{r',j}(p)$ for $p : leaf_{r'}(p) = leaf_{r'}(i), \sigma_{r'}(p) \le j)$ for $r' = 1..s$, $i = 1..n, j \ge \sigma_{r'}(i) - 1$;
return T, M

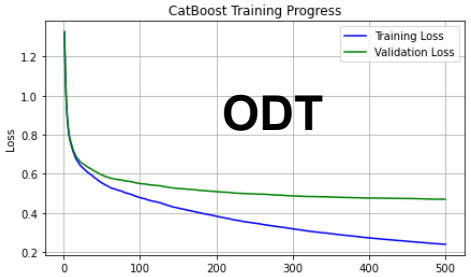
- Особенности:
  - «прогноз» в листьях (а значит и всего дерева) не константа для всех примеров листа, а вектор длины l с усреднением прогноза для x_i с учетом его листа и порядка в X^{jr}
  - Потери (критерий оценки разбиения) – косинусная мера сходства с вектором несмещенных антиградиентов
  - Пересчет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

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

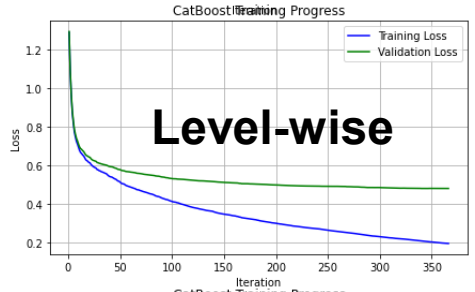
Пример CatBoost с ordered boosting

```python
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 consequtive rounds
                           loss_function='MultiClass')  # used for Multiclass classification tasks
```

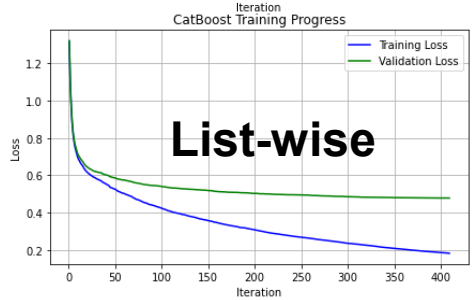
```
0:      learn: 1.3244454        test: 1.3274746 best: 1.3274746 (0)     total: 10.8ms   remaining: 5.39s
100:    learn: 0.4770844        test: 0.5501835 best: 0.5501835 (100)   total: 1.25s    remaining: 4.95s
200:    learn: 0.3820112        test: 0.5085822 best: 0.5085822 (200)   total: 2.53s    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 consequtive 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.
```

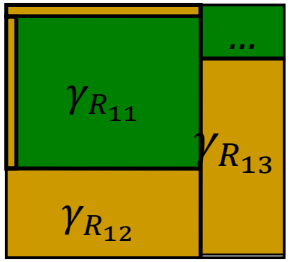
Упорядоченный

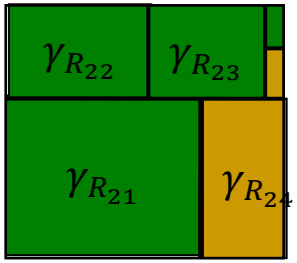
Особенности классического градиентного бустинга деревьев решений

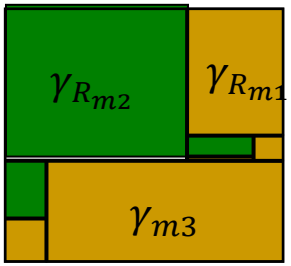
- В сравнении с основным конкурентом случайным лесом:
  - уменьшает не только разброс, но и **смещение** всего ансамбля
  - с ростом числа базовых моделей не склонен к **переобучению**, когда нет шума (выбросов), но склонен когда **выбросы** есть
  - **плохо распараллеливается** (только на уровне отдельных 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 **бутсретпинг** 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: $F_m(x) = F_0 + \alpha_1 T_1(x) + \alpha_2 T_2(x) + \ldots + \alpha_m T_m(x)$

$F_m(x) = F_0 + \alpha_1 *$ 

$+\alpha_2 *$ 

$+ \cdots + \alpha_m *$ 

  - 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:

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

$$\approx \sum_i L\big(y_i, F_{m-1}(x_i)\big) + b(x_i)\frac{\partial L(y_i, F_{m-1})}{\partial F_{m-1}}(x_i) + \frac{1}{2}b^2(x_i)\frac{\partial^2 L(y_i, F_{m-1})}{(\partial F_{m-1})^2}(x_i) \ldots$$

  - 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:

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

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

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

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:

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

  - 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
```

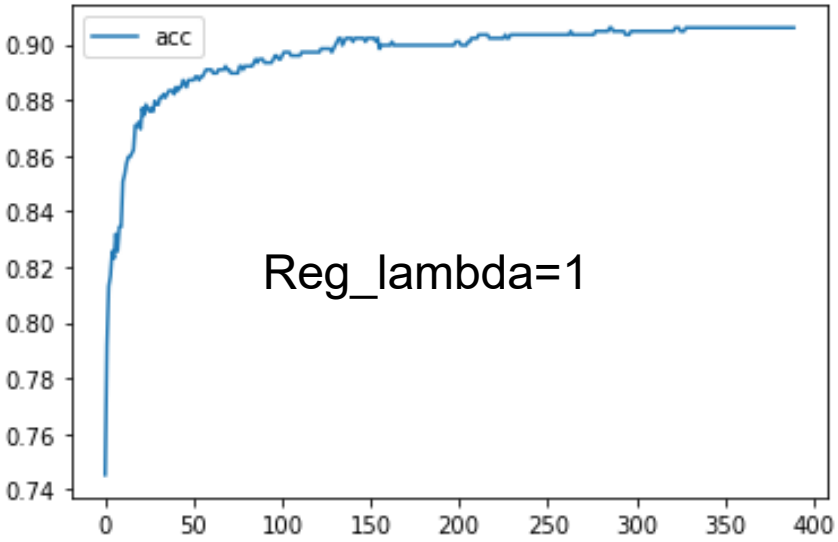
```
# https://xgboost.readthedocs.io/en/stable/parameter.html
param = dict(objective="multi:softprob", num_class=10, # softmax
             booster="gbtree", learning_rate=0.1, max_depth=5, subsample=0.5,
             alpha=5, # l1 regularization
             tree_method="auto") # approx, hist, gpu_hist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

```
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')]
```

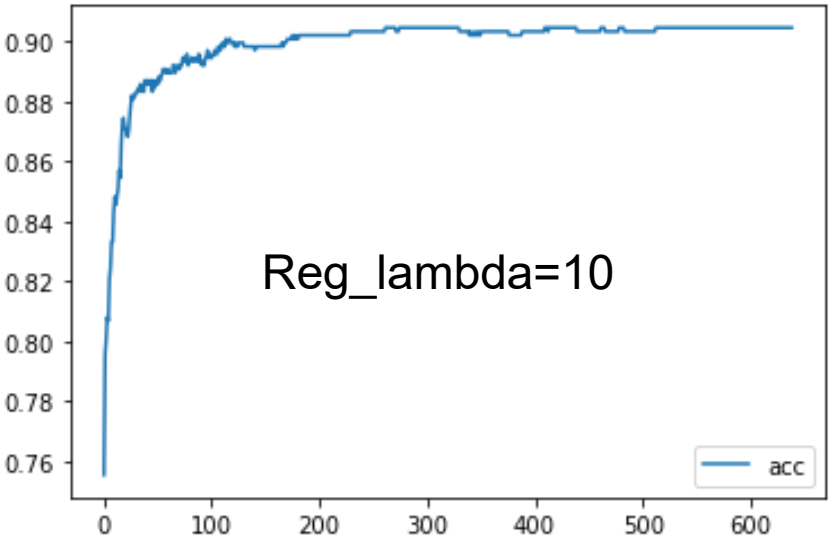
```
xgb_classifier = xgb.train(param, dtrain, 1000, evals=evallist, early_stopping_rounds=10)
# xgb_classifier.save_model('mymodel')
```

```
[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



Reg_lambda=1



Reg_lambda=10

```
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.

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
- ECOC, 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