

Градиентный бустинг

Задача восстановления зависимости $y: X \rightarrow Y$ по точкам обучающей выборки (x_i, y_i) , $y_i = y(x_i)$, $i = 1, \dots, \ell$.

Определение

Линейной композицией базовых алгоритмов $a_t(x) = C(b_t(x))$, $t = 1, \dots, T$, называется суперпозиция функций

$$a(x) = C\left(\sum_{t=1}^T \alpha_t b_t(x)\right),$$

где $C: \mathbb{R} \rightarrow Y$ — решающее правило, $\alpha_t \geq 0$.

- **Пример 1:** классификация на 2 класса, $Y = \{-1, +1\}$;
 $C(b) = \text{sign}(b)$, $a(x) = \text{sign}(b(x))$,
 $b: X \rightarrow \mathbb{R}$ — дискриминантная функция.
- **Пример 2:** регрессия, $Y = \mathbb{R}$,
 $C(b) = b$, $a(x) = b(x)$, решающее правило не используется.

Линейная композиция базовых алгоритмов:

$$b(x) = \sum_{t=1}^T \alpha_t b_t(x), \quad x \in X, \quad \alpha_t \in \mathbb{R}_+.$$

Функционал качества с произвольной функцией потерь $\mathcal{L}(b, y)$:

$$Q(\alpha, b) = \sum_{i=1}^{\ell} \mathcal{L} \left(\underbrace{\sum_{t=1}^{T-1} \alpha_t b_t(x_i)}_{u_{T-1,i}} + \alpha b(x_i), y_i \right) \rightarrow \min_{\alpha, b}.$$

$\underbrace{\hspace{10em}}_{u_{T,i}}$

Ищем вектор $u = (b(x_i))_{i=1}^{\ell}$ из R^{ℓ} , минимизирующий $Q(\alpha, b)$.

$u_{T-1} = (u_{T-1,i})_{i=1}^{\ell}$ — текущее приближение вектора u

$u_T = (u_{T,i})_{i=1}^{\ell}$ — следующее приближение вектора u

- Градиентный метод минимизации $Q(u) \rightarrow \min, u \in \mathbb{R}^\ell$:

$u_0 :=$ начальное приближение;

$$u_{T,i} := u_{T-1,i} - \alpha g_i, \quad i = 1, \dots, \ell;$$

$g_i = \mathcal{L}'(u_{T-1,i}, y_i)$ — компоненты вектора градиента,
 α — градиентный шаг.

- Добавление базового алгоритма b_T :

$$u_{T,i} := u_{T-1,i} + \alpha b_T(x_i), \quad i = 1, \dots, \ell$$

Будем искать такой базовый алгоритм b_T , чтобы вектор $(b_T(x_i))_{i=1}^\ell$ приближал вектор антиградиента $(-g_i)_{i=1}^\ell$:

$$b_T := \arg \max_b \sum_{i=1}^{\ell} (b(x_i) + g_i)^2$$

Вход: обучающая выборка X^ℓ ; **параметр** T ;

Выход: базовые алгоритмы и их веса $\alpha_t b_t$, $t = 1, \dots, T$;

1: инициализация: $u_i := 0$, $i = 1, \dots, \ell$;

2: **для всех** $t = 1, \dots, T$

3: найти базовый алгоритм, приближающий градиент:

$$b_t := \arg \min_b \sum_{i=1}^{\ell} (b(x_i) + \mathcal{L}'(u_i, y_i))^2;$$

4: решить задачу одномерной минимизации:

$$\alpha_t := \arg \min_{\alpha > 0} \sum_{i=1}^{\ell} \mathcal{L}(u_i + \alpha b_t(x_i), y_i);$$

5: обновить значения композиции на объектах выборки:

$$u_i := u_i + \alpha_t b_t(x_i); \quad i = 1, \dots, \ell;$$

Известно, что рандомизации могут повышать качество композиции за счёт повышения различности базовых алгоритмов (на этом основаны bagging, RF, RSM)

Идея:

на шагах 3–5 использовать не всю выборку X^ℓ ,
а случайную подвыборку с повторениями, как в бэггинге.

Преимущества:

- улучшается качество
- улучшается сходимость
- уменьшается время обучения

Исторически первый вариант бустинга (1995).

Задача классификации на два класса, $Y = \{-1, +1\}$,

$\mathcal{L}(b(x_i), y_i) = e^{-b(x_i)y_i}$ — экспоненциальная функция потерь, убывающая функция отступа $M_i = b(x_i)y_i$

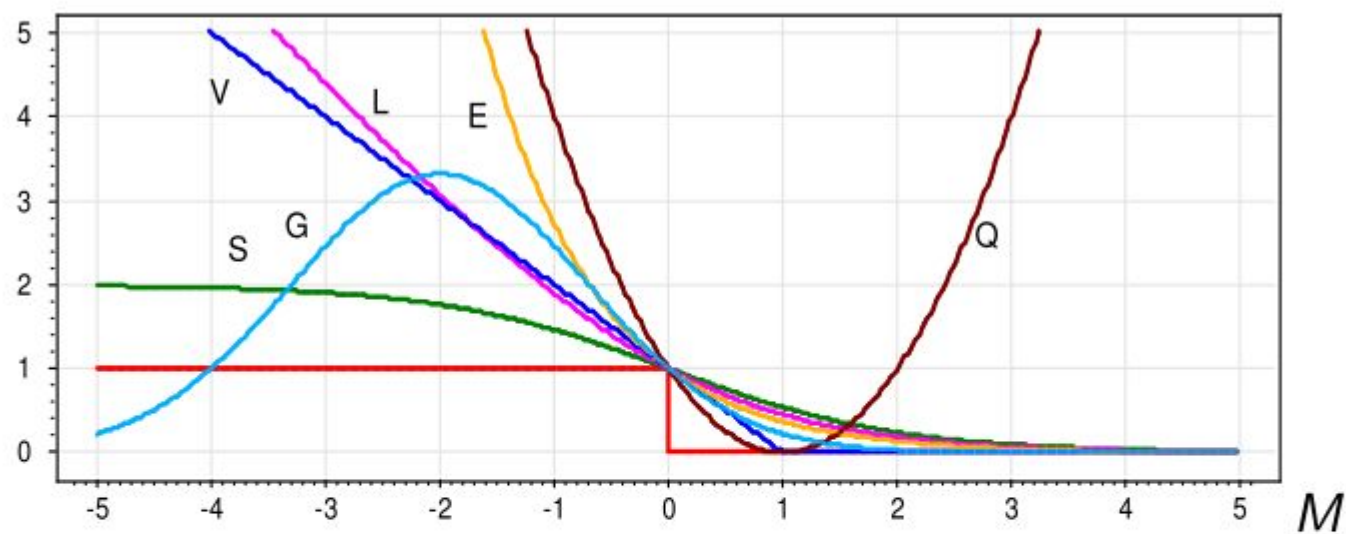
Преимущества:

- для обучения b_t на каждом шаге t решается стандартная задача минимизации взвешенного эмпирического риска
- задача оптимизации α_t решается аналитически

Недостаток:

- AdaBoost слишком чувствителен к выбросам из-за экспоненциального роста функции потерь при $M_i < 0$

Функции потерь $\mathcal{L}(M)$ в задачах классификации на два класса



$E(M) = e^{-M}$ — экспоненциальная (AdaBoost);

$L(M) = \log_2(1 + e^{-M})$ — логарифмическая (LogitBoost);

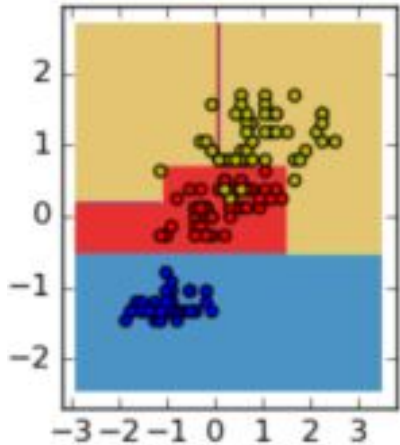
$G(M) = \exp(-cM(M + s))$ — гауссовская (BrownBoost);

$Q(M) = (1 - M)^2$ — квадратичная;

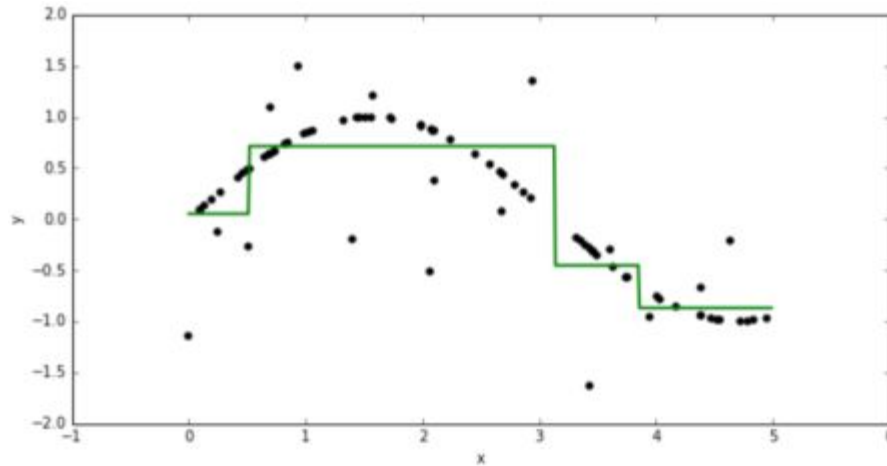
$S(M) = 2(1 + e^M)^{-1}$ — сигмоидная;

$V(M) = (1 - M)_+$ — кусочно-линейная (SVM);

Градиентный бустинг над решающими деревьями



Классификация



Регрессия

Решающее дерево — это кусочно-постоянная функция:

$$b(x) = \sum_{t=1}^T \alpha_t [x \in \Omega_t],$$

где T — число листьев,
 Ω_t — область t -го листа,
 α_t — прогноз в t -м листе.

Решающее дерево разбивает пространство объектов на области $\Omega_1, \dots, \Omega_T$.

•

$$u_N(x) = u_{N-1}(x) + b(x)$$

$$b(x) = \sum_{t=1}^T \alpha_t [x \in \Omega_t]$$

$$u_N(x) = u_{N-1}(x) + \sum_{t=1}^T \alpha_t [x \in \Omega_t]$$

- $$u_N(x) = u_{N-1}(x) + \sum_{t=1}^T \alpha_t [x \in \Omega_t]$$

$$\sum_{i=1}^l L \left(y_i, u_{N-1}(x) + \sum_{t=1}^T \alpha_t [x \in \Omega_t] \right) \rightarrow \min_{\alpha_1, \dots, \alpha_T}$$

$$\alpha_t = \arg \min_{\alpha > 0} \underbrace{\sum_{x_i \in \Omega_t} \mathcal{L}(u_{t-1,i} + \alpha, y_i)}_{\text{суммарная потеря в } t\text{-м листе}}.$$

Оптимизация прогнозов в листьях:

$$\alpha_t = \arg \min_{\alpha > 0} \sum_{x_i \in \Omega_t} \mathcal{L}(u_{t-1,i} + \alpha, y_i).$$

Для некоторых функций потерь решение находится аналитически:

- средний квадрат ошибок, MSE, $\mathcal{L}(b, y) = (b - y)^2$:

$$\alpha_t = \frac{1}{|\Omega_t|} \sum_{x_i \in \Omega_t} (y_i - u_{t-1,i}).$$

- средняя абсолютная ошибка, MAE, $\mathcal{L}(b, y) = |b - y|$:

$$\alpha_t = \operatorname{median}_{x_i \in \Omega_t} \{y_i - u_{t-1,i}\}.$$

В общем случае аналитического решения нет.

- Градиентный бустинг — наиболее общий из всех бустингов:
 - произвольная функция потерь
 - произвольное пространство оценок R
 - подходит для регрессии, классификации, ранжирования
- Важное открытие середины 90-х: обобщающая способность бустинга не ухудшается с ростом сложности T
- Стохастический вариант SGB — лучше и быстрее
- Градиентный бустинг над решающими деревьями часто работает лучше, чем случайный лес
- Технология Yandex.MatrixNet — это градиентный бустинг над «небрежными» решающими деревьями ODT (ODT — oblivious decision tree)

Вопрос

Для чего отдельные деревья в случайном лесу обучаются по подвыборкам объектов? Выберите основную причину.

- Чтобы деревья не были одинаковыми
- Чтобы деревья не были переобученными
- Чтобы сократить время обучения

Вопрос

Какие из этих утверждений относятся к методу градиентного бустинга над решающими деревьями?

- Деревья строятся независимо

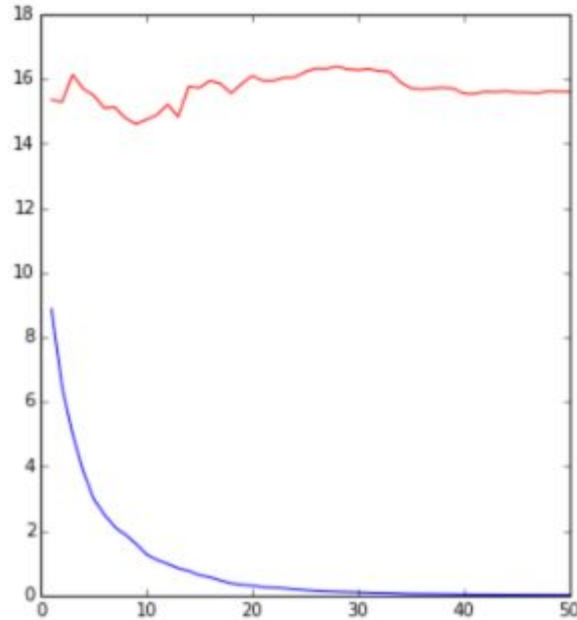
- Обучение отдельных деревьев на подвыборках хорошо сказывается на качестве композиции

- Деревья, как правило, делают небольшой глубины

- Деревья строятся последовательно, обучение следующего дерева зависит от ошибок уже построенной композиции

- Деревья, как правило, делают глубокими

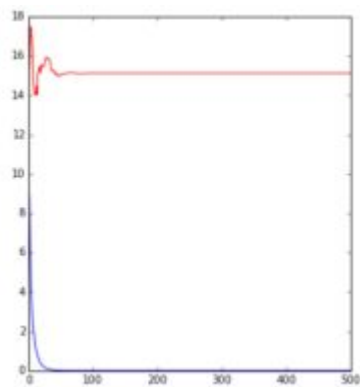
Ошибка в зависимости от числа деревьев



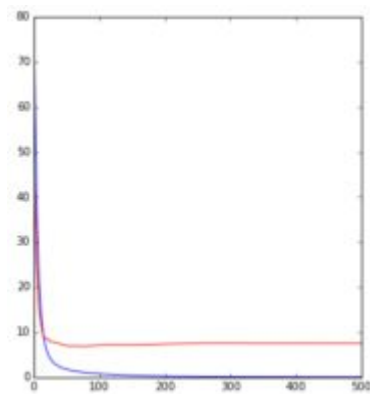
$$u_{T,i} := u_{T-1,i} + \alpha b_T(x_i), \quad i = 1, \dots, \ell$$

$$\alpha \in (0; 1]$$

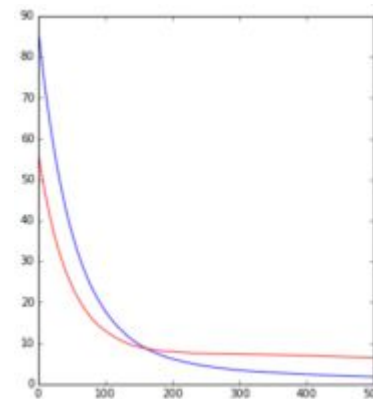
Качество бустинга



$\alpha=1$

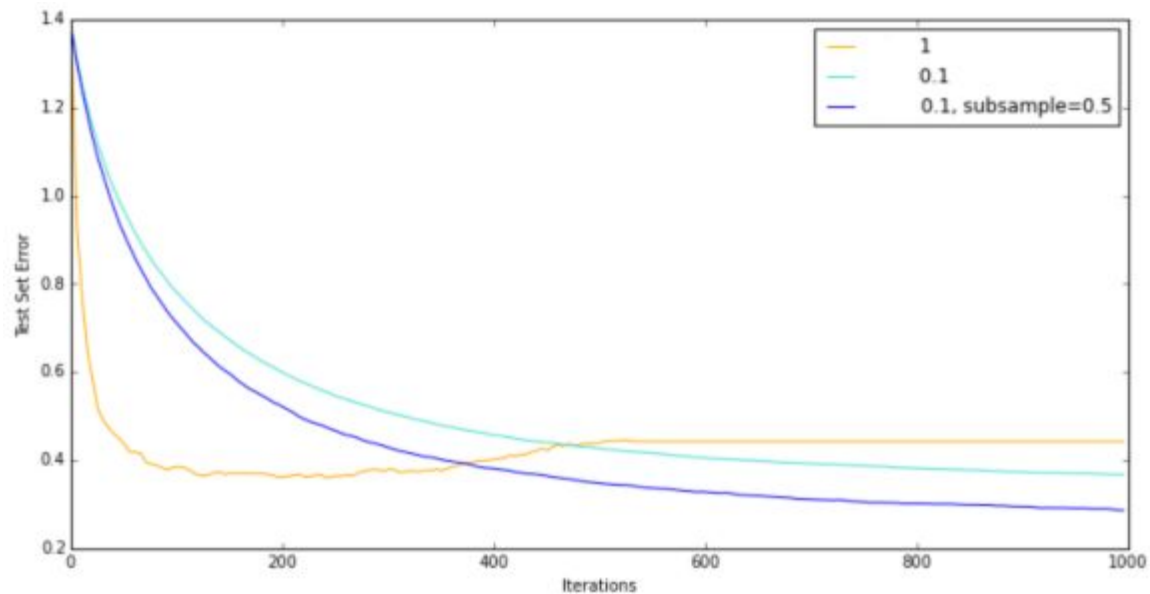


$\alpha=0.1$



$\alpha=0.01$

Стохастический градиентный бустинг



Gradient Boosting for classification.

GB builds an additive model in a forward stage-wise fashion; it allows for the optimization of arbitrary differentiable loss functions. In each stage `n_classes` regression trees are fit on the negative gradient of the binomial or multinomial deviance loss function. Binary classification is a special case where only a single regression tree is induced.

Read more in the [User Guide](#).

Parameters: `loss` : {'deviance', 'exponential'}, optional (default='deviance')

loss function to be optimized. 'deviance' refers to deviance (= logistic regression) for classification with probabilistic outputs. For loss 'exponential' gradient boosting recovers the AdaBoost algorithm.

`learning_rate` : float, optional (default=0.1)

learning rate shrinks the contribution of each tree by `learning_rate`. There is a trade-off between `learning_rate` and `n_estimators`.

`n_estimators` : int (default=100)

The number of boosting stages to perform. Gradient boosting is fairly robust to overfitting so a large number usually results in better performance.

`max_depth` : integer, optional (default=3)

maximum depth of the individual regression estimators. The maximum depth limits the number of nodes in the tree. Tune this parameter for best performance; the best value depends on the interaction of the input variables.

`criterion` : string, optional (default="friedman_mse")

The function to measure the quality of a split. Supported criteria are "friedman_mse" for the mean squared error with improvement score by Friedman, "mse" for mean squared error, and "mae" for the mean absolute error. The default value of "friedman_mse" is generally the best as it can provide a better approximation in some cases.

New in version 0.18.

min_samples_split : int, float, optional (default=2)

The minimum number of samples required to split an internal node:

- If int, then consider min_samples_split as the minimum number.
- If float, then min_samples_split is a percentage and $\text{ceil}(\text{min_samples_split} * \text{n_samples})$ are the minimum number of samples for each split.

Changed in version 0.18: Added float values for percentages.

min_samples_leaf : int, float, optional (default=1)

The minimum number of samples required to be at a leaf node:

- If int, then consider min_samples_leaf as the minimum number.
- If float, then min_samples_leaf is a percentage and $\text{ceil}(\text{min_samples_leaf} * \text{n_samples})$ are the minimum number of samples for each node.

Changed in version 0.18: Added float values for percentages.

min_weight_fraction_leaf : float, optional (default=0.)

The minimum weighted fraction of the sum total of weights (of all the input samples) required to be at a leaf node. Samples have equal weight when sample_weight is not provided.

subsample : float, optional (default=1.0)

The fraction of samples to be used for fitting the individual base learners. If smaller than 1.0 this results in Stochastic Gradient Boosting. subsample interacts with the parameter n_estimators. Choosing subsample < 1.0 leads to a reduction of variance and an increase in bias.

max_features : int, float, string or None, optional (default=None)

The number of features to consider when looking for the best split:

- If int, then consider max_features features at each split.
- If float, then max_features is a percentage and $\text{int}(\text{max_features} * \text{n_features})$ features are considered at each split.
- If "auto", then $\text{max_features} = \text{sqrt}(\text{n_features})$.
- If "sqrt", then $\text{max_features} = \text{sqrt}(\text{n_features})$.
- If "log2", then $\text{max_features} = \text{log}_2(\text{n_features})$.
- If None, then $\text{max_features} = \text{n_features}$.

Choosing $\text{max_features} < \text{n_features}$ leads to a reduction of variance and an increase in bias.

Note: the search for a split does not stop until at least one valid partition of the node samples is found, even if it requires to effectively inspect more than max_features features.

max_leaf_nodes : int or None, optional (default=None)

Grow trees with max_leaf_nodes in best-first fashion. Best nodes are defined as relative reduction in impurity. If None then unlimited number of leaf nodes.

min_impurity_split : float,

Threshold for early stopping in tree growth. A node will split if its impurity is above the threshold, otherwise it is a leaf.

Deprecated since version 0.19: min_impurity_split has been deprecated in favor of min_impurity_decrease in 0.19 and will be removed in 0.21. Use min_impurity_decrease instead.

min_impurity_decrease : float, optional (default=0.)

A node will be split if this split induces a decrease of the impurity greater than or equal to this value.

The weighted impurity decrease equation is the following:

$$N_t / N * (\text{impurity} - N_{t_R} / N_t * \text{right_impurity} - N_{t_L} / N_t * \text{left_impurity})$$

where N is the total number of samples, N_t is the number of samples at the current node, N_{t_L} is the number of samples in the left child, and N_{t_R} is the number of samples in the right child.

N , N_t , N_{t_R} and N_{t_L} all refer to the weighted sum, if `sample_weight` is passed.

New in version 0.19.

init : BaseEstimator, None, optional (default=None)

An estimator object that is used to compute the initial predictions. `init` has to provide `fit` and `predict`. If None it uses `loss.init_estimator`.

verbose : int, default: 0

Enable verbose output. If 1 then it prints progress and performance once in a while (the more trees the lower the frequency). If greater than 1 then it prints progress and performance for every tree.

warm_start : bool, default: False

When set to `True`, reuse the solution of the previous call to fit and add more estimators to the ensemble, otherwise, just erase the previous solution.

random_state : int, RandomState instance or None, optional (default=None)

If int, `random_state` is the seed used by the random number generator; If RandomState instance, `random_state` is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`.