Урок 6. Градиентный бустинг. AdaBoost

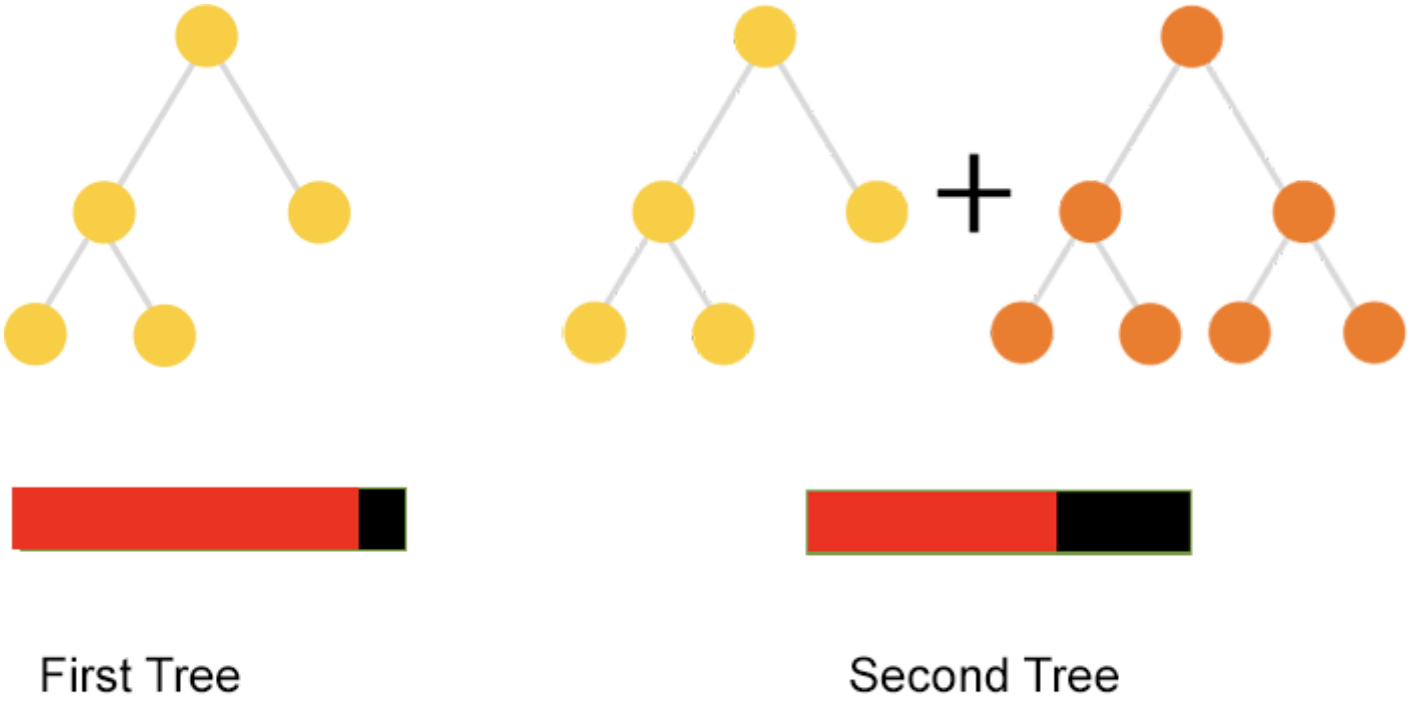
План занятия

Теоретическая часть

В этом уроке мы продолжаем тему ансамблей алгоритмов, рассматривая еще один их вид - *градиентный бустинг*.

Вспоминая тему предыдущего урока, случайные леса, напомним, что случайный лес - это ансамбль деревьев небольшой глубины, строящихся независимо друг от друга. В независимости построения деревьев кроется и **плюс и минус алгоритма**: с одной стороны, построение деревьев можно **распараллеливать** и, например, организовывать на разных ядрах процессора, с другой стороны, следствием их независимости является тот факт, что для решения сложных задач требуется очень **большое количество деревьев**. В этих случаях случаях (при большой выборке или большом количестве признаков) обучение случайного леса может требовать очень много ресурсов, а если для ограничения их потребления слишком ограничивать глубину деревьев, они могут не уловить все закономерности в данных и иметь большой сдвиг (и, следовательно, ошибку).

Бустинг является своеобразным решением этой проблемы: он заключается в **последовательном** построении ансамбля, когда деревья строятся одно за другим, и при этом каждое следующее дерево строится таким образом, чтобы исправлять ошибки уже построенного на данный момент ансамбля. При таком подходе базовые алгоритмы могут быть достаточно простыми, то есть можно использовать неглубокие деревья.



Градиентный бустинг (GBM)

Видео (https://youtu.be/sDv4f4s2SB8) с подробным объяснением алгоритма

Продемонстрируем работу бустинга

Ввод [1]:

```
from sklearn.datasets import load_diabetes
from sklearn.tree import DecisionTreeRegressor
from sklearn.metrics import r2_score


X, y = load_diabetes(return_X_y=True, as_frame=True)
X
```

Out[1]:

	age	sex	bmi	bp	s1	s2	s3	s4	s5	s6
0	0.038076	0.050680	0.061696	0.021872	-0.044223	-0.034821	-0.043401	-0.002592	0.019908	-0.017646
1	-0.001882	-0.044642	-0.051474	-0.026328	-0.008449	-0.019163	0.074412	-0.039493	-0.068330	-0.092204
2	0.085299	0.050680	0.044451	-0.005671	-0.045599	-0.034194	-0.032356	-0.002592	0.002864	-0.025930
3	-0.089063	-0.044642	-0.011595	-0.036656	0.012191	0.024991	-0.036038	0.034309	0.022692	-0.009362
4	0.005383	-0.044642	-0.036385	0.021872	0.003935	0.015596	0.008142	-0.002592	-0.031991	-0.046641
...	...	...	...	...	...	...	...	...	...	...
437	0.041708	0.050680	0.019662	0.059744	-0.005697	-0.002566	-0.028674	-0.002592	0.031193	0.007207
438	-0.005515	0.050680	-0.015906	-0.067642	0.049341	0.079165	-0.028674	0.034309	-0.018118	0.044485
439	0.041708	0.050680	-0.015906	0.017282	-0.037344	-0.013840	-0.024993	-0.011080	-0.046879	0.015491
440	-0.045472	-0.044642	0.039062	0.001215	0.016318	0.015283	-0.028674	0.026560	0.044528	-0.025930
441	-0.045472	-0.044642	-0.073030	-0.081414	0.083740	0.027809	0.173816	-0.039493	-0.004220	0.003064

442 rows × 10 columns

Ввод [2]:
```
y
```

Out[2]:
```
0      151.0
1       75.0
2      141.0
3      206.0
4      135.0
       ...
437    178.0
438    104.0
439    132.0
440    220.0
441     57.0
Name: target, Length: 442, dtype: float64
```

Ввод [3]:
```
# Обучаем первое дерево
tree1 = DecisionTreeRegressor(max_depth=3,
                              random_state=2)
tree1.fit(X, y)

prediction1 = tree1.predict(X)
print(f'R2 score {r2_score(y, prediction1)}')
```

```
R2 score 0.5006720154703376
```

Ввод [4]:
```
# Подсчитываем остатки
residual1 = y - prediction1
y[2], prediction1[2], residual1[2]
```

Out[4]:
```
(141.0, 208.57142857142858, -67.57142857142858)
```

Ввод [5]:
```
# Обучаем второе дерево на ошибках предыдущих
tree2 = DecisionTreeRegressor(max_depth=3,
                              random_state=2)

tree2.fit(X, residual1)

prediction2 = tree1.predict(X) + tree2.predict(X)
print(f'R2 score {r2_score(y, prediction2)}')
```

```
R2 score 0.5785866108916171
```

Ввод [6]:
```
# Подсчитываем остатки
residual2 = y - prediction2
y[2], prediction2[2], residual2[2]
```

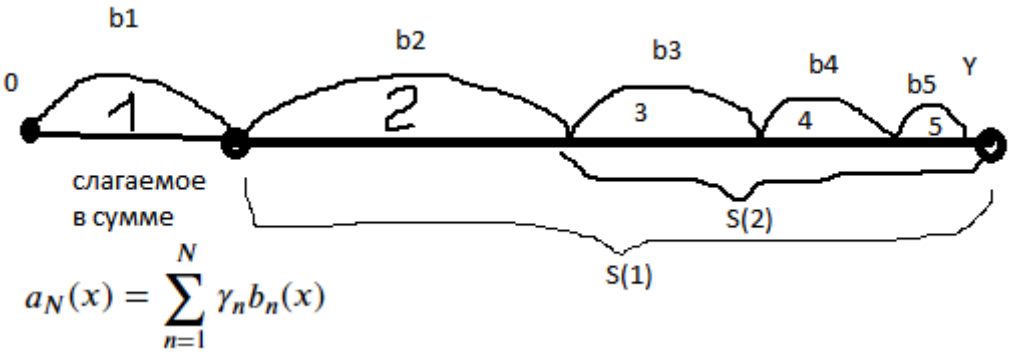
Out[6]:
```
(141.0, 207.5052553799773, -66.50525537997731)
```

Ввод [7]:
```
# Обучаем третье дерево на ошибках предыдущих
tree3 = DecisionTreeRegressor(max_depth=3,
                              random_state=2)

tree3.fit(X, residual2)

prediction3 = tree1.predict(X) + tree2.predict(X) + tree3.predict(X)
print(f'R2 score {r2_score(y, prediction3)}')
```

```
R2 score 0.6271203539706915
```

Ввод [8]:
```
# Подсчитываем остатки
residual3 = y - prediction3
y[2], prediction3[2], residual3[2]
```

Out[8]:
```
(141.0, 174.76070684814292, -33.76070684814292)
```



Алгоритм построения градиентного бустинга

1. Инициализация начального алгоритма $b_0(x)$
2. Цикл по n = 1,2,3,...:

- Подсчитывание остатков $s = \left(-\frac{\partial L}{\partial z}\Big|_{z=a_{n-1}(x_1)}, \ldots, -\frac{\partial L}{\partial z}\Big|_{z=a_{n-1}(x_l)} \right)$;
- Обучение нового алгоритма $b_n(x) = \underset{s}{\operatorname{argmin}} \frac{1}{l} \sum_{i=1}^{l} (b(x_i) - s_i)^2$;
- Добавление алгоритма в композицию $a_n(x) = a_{n-1}(x) + \eta b_n(x)$.

Итоговый алгоритм ищется в виде взвешенной суммы базовых алгоритмов (обратите внимание: не среднего, а суммы):

$$a_N(x) = \sum_{n=1}^{N} b_n(x).$$

В случае регрессии задача состоит в минимизации среднеквадратичного функционала ошибки:

$$\frac{1}{l} \sum_{i=1}^{l} (a(x_i) - y_i)^2 \to \min.$$

Так как ансамбль строится итеративно, нужно в начале обучить первый простой алгоритм:

$$b_1(x) = \underset{b}{\operatorname{argmin}} \frac{1}{l} \sum_{i=1}^{l} (b(x_i) - y_i)^2.$$

После того, как мы нашли первый алгоритм $b_1(x)$, нам нужно добавить в ансамбль еще один алгоритм $b_2(x)$. Для начала найдем разницу ответов первого алгоритма с реальными ответами:

$$s_i^{(1)} = y_i - b_1(x_i).$$

Если прибавить эти значения к полученным предсказаниям, получим идеальный ответ. Таким образом, новый алгоритм логично обучать так, чтобы его ответы были максимально близки к этой разнице, чтобы при их прибавлении к ответам первого алгоритма мы получили близкие к реальным. Значит, второй алгоритм будет обучаться на следующем функционале ошибки:

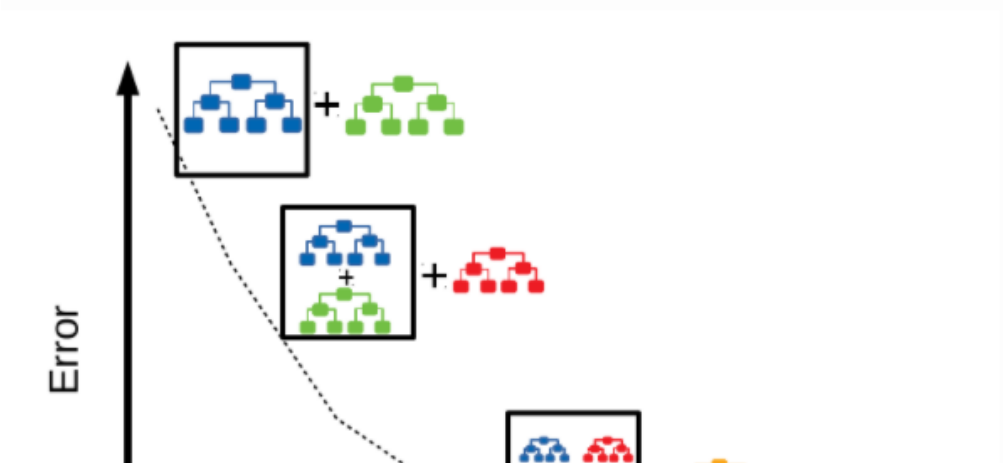
$$b_2(x) = \underset{b}{\operatorname{argmin}} \frac{1}{l} \sum_{i=1}^{l} (b(x_i) - s_i^{(1)})^2 = \underset{b}{\operatorname{argmin}} \frac{1}{l} \sum_{i=1}^{l} (b(x_i) - (y_i - b_1(x_i)))^2.$$

Каждый следующий алгоритм также настраивается на остатки композиции из предыдущих алгоритмов:

$$b_N(x) = \underset{b}{\operatorname{argmin}} \frac{1}{l} \sum_{i=1}^{l} (b(x_i) - s_i^{(N)})^2,$$

$$s_i^{(N)} = y_i - \sum_{n=1}^{N-1} b_n(x_i) = y_i - a_{N-1}(x_i).$$

Таким образом, каждый новый алгоритм корректирует ошибки предыдущих, и так продолжается до момента получения приемлемой ошибки на композиции. Вектор коэффициентов s при этом называют *вектором сдвига*.



Выбор сдвига из условия $s_i = y_i - a_{N-1}(x_i)$ требует точного совпадения полученных предсказаний и ответов, однако, в более общем случае вектор сдвига принимают с учетом особенностей используемой в данном случае функции потерь: вектор сдвига должен ее минимизировать, то есть направлять в сторону уменьшения. Как мы помним из метода градиентного спуска, направление наискорейшего убывания функции совпадает с ее антиградиентом. Таким образом, если при обучении мы минимизируем функционал ошибки $L(y, z)$

$$\sum_{i=1}^{l} L(y_i, a_{N-1}(x_i) + s_i) \to \min_s,$$

сдвиг на каждом шаге должен быть противоположен производной функции потерь в точке $z = a_{N-1}(x_i)$.

$$s_i = -\frac{\partial L}{\partial z}\bigg|_{z=a_{N-1}(x_i)}.$$

Каждый новый алгоритм таким образом выбирается так, чтобы как можно лучше приближать антиградиент ошибки на обучающей выборке.

После того, как мы вычислили требуемый для минимизации ошибки сдвиг s, нужно настроить алгоритм $b_N(x)$ так, чтобы он давал максимально близкие к нему ответы, то есть обучать его именно на вектор сдвига. Близость ответов алгоритма к сдвигу обычно оценивается с помощью среднеквадратичной ошибки независимо от условий исходной задачи (так как исходно используемая функция потерь L уже учтена в сдвигах s_i):

$$b_N(x) = \operatorname{argmin} \frac{1}{l} \sum (b(x_i) - s_i)^2$$

Обычно в качестве функции потерь в задачах регрессии принимается *квадратичная функция потерь* (L_2 loss):

$$L(y, z) = (y - z)^2,$$

его **производная** по z примет вид

$$L'(y, z) = 2(z - y)$$

или модуль отклонения (L_1 loss)

$$L(y, z) = |y - z|,$$

его **производная** по z будет иметь вид

$$L'(y, z) = \text{sign}(z - y).$$

В случае классификации - логистическая функция потерь (метки -1, +1), где z - оценка принадлежности классу:

$$L(y, z) = log(1 + exp(-yz))$$

ее **производная**:

$$L'(y, z) = -\frac{y \cdot exp(-yz)}{1 + exp(-yz)}.$$

Следует помнить, что компоненты s, вычисляемые через эти производные, берутся с минусом.

Аналогично алгоритму градиентного спуска, имеет смысл добавлять ответ каждого нового алгоритма не полностью, а с некоторым шагом $\eta \in (0, 1]$, так как базовые алгоритмы обычно достаточно простые (например, деревья малой глубины), и они могут плохо приближать вектор антиградиента, и тогда вместо приближения к минимуму мы будем получать случайное блуждание в пространстве. В градиентном бустинге такой прием называется сокращением шага.

$$a_N(x) = a_{N-1}(x) + \eta b_N(x).$$

Градиентный бустинг склонен к переобучению при увеличении числа итераций N или глубины входящих в него деревьев. Стоит об этом помнить при построении алгоритма и выбирать оптимальные параметры по отложенной выборке или с помощью кросс-валидации.

Алгоритм построения модели градиентного бустинга

1. Для инициализации выбирается произвольный простой алгоритм $b_0(x)$, в его роли можно брать обычные константные алгоритмы: в случае задачи регрессии это может быть

$$b_0(x) = 0$$

или среднее значение по всем объектам обучающей выборки

$$b_0(x) = \frac{1}{l}\sum_{i=1}^{l} y_i;$$

в случае классификации - самый часто встречающийся в выборке класс

$$b_0(x) = \underset{y}{\operatorname{argmax}} \sum_{i=1}^{l} [y_i = y].$$

2. Для каждой итерации вычисляется вектор сдвига s:

$$s = \left(-\frac{\partial L}{\partial z}\bigg|_{z=a_{n-1}(x_1)}, \ldots, -\frac{\partial L}{\partial z}\bigg|_{z=a_{n-1}(x_l)} \right);$$

находится алгоритм

$$b_n(x) = \underset{s}{\operatorname{argmin}} \frac{1}{l}\sum_{i=1}^{l} (b(x_i) - s_i)^2;$$

и добавляется в имеющийся ансамбль с умножением на шаг η, называемый *скоростью обучения* (опционно)

$$a_n(x) = a_{n-1}(x) + \eta b_n(x).$$

3. При достижении критериев остановки компонуется итоговая модель.

Стохастический градиентный бустинг

Как и в случае с градиентым спуском, есть так называемый стохастический градиентный бустинг, являющийся упрощенной (в плане потребления ресурсов) версией алгоритма. Его суть заключается в обучении каждого нового базового алгоритма на новой итерации не на всей обучающей выборке, а на некоторой ее случайной подвыборке. Практика показывает, что такой алгоритм позволяет получить такую же ошибку или даже уменьшить ее при том же числе итераций, что и в случае использования обычного бустинга.

Реализация алгоритма градиентного бустинга

Реализуем средствами Python алгоритм градиентного бустинга для деревьев решений.

Реализация деревьев решений была дважды продемонстрирована в предыдущих уроках, в этом не будем ее повторять и возьмем готовую реализацию дерева решений для регрессии из библиотеки `sklearn` .

Ввод [9]:
```
from sklearn import model_selection
from sklearn.tree import DecisionTreeRegressor
from sklearn.datasets import load_diabetes
import matplotlib.pyplot as plt
import numpy as np
```

Используем один из "игрушечных" датасетов из той же библиотеки.

Ввод [10]:
```
X, y = load_diabetes(return_X_y=True)
X.shape, y.shape
```

Out[10]:
```
((442, 10), (442,))
```

Разделим выборку на обучающую и тестовую в соотношении 75/25.

Ввод [11]:
```
X_train, X_test, y_train, y_test = model_selection.train_test_split(X, y, test_size=0.25)
```

Напишем функцию, реализующую предсказание в градиентном бустинге.

Ввод [12]:
```
def gb_predict(X, trees_list, eta):
    # Реализуемый алгоритм градиентного бустинга будет инициализироваться нулевыми значениями,
    # поэтому все деревья из списка trees_list уже являются дополнительными и при предсказании
    # прибавляются с шагом eta

#     predictions = np.zeros(X.shape[0])
#     for i, x in enumerate(X):
#         prediction = 0
#         for alg in trees_list:
#             prediction += eta * alg.predict([x])[0]
#         predictions[i] = prediction

    predictions = np.array(
        [sum([eta * alg.predict([x])[0] for alg in trees_list]) for x in X]
    )

    return predictions
```

В качестве функционала ошибки будем использовать среднеквадратичную ошибку. Реализуем соответствующую функцию.

Ввод [13]:
```
def mean_squared_error(y_real, prediction):
    return (sum((y_real - prediction)**2)) / len(y_real)
```

Используем L_2 loss $L(y, z) = (y - z)^2$, ее производная по z примет вид $L'(y, z) = 2(z - y)$. Реализуем ее также в виде функции (коэффициент 2 можно отбросить).

Ввод [14]:
```
def residual(y, z):
    return - (z - y)
```

Реализуем функцию обучения градиентного бустинга.

Ввод [15]:

```
def gb_fit(n_trees, max_depth, X_train, X_test, y_train, y_test, eta):

    # Деревья будем записывать в список
    trees = []

    # Будем записывать ошибки на обучающей и тестовой выборке на каждой итерации в список
    train_errors = []
    test_errors = []

    for i in range(n_trees):
        tree = DecisionTreeRegressor(max_depth=max_depth, random_state=42)

        # первый алгоритм просто обучаем на выборке и добавляем в список
        if len(trees) == 0:
            # обучаем первое дерево на обучающей выборке
            tree.fit(X_train, y_train)

            train_errors.append(mean_squared_error(y_train, gb_predict(X_train, trees, eta)))
            test_errors.append(mean_squared_error(y_test, gb_predict(X_test, trees, eta)))
        else:
            # Получим ответы на текущей композиции
            target = gb_predict(X_train, trees, eta)

            # алгоритмы начиная со второго обучаем на сдвиг
            tree.fit(X_train, residual(y_train, target))

            train_errors.append(mean_squared_error(y_train, gb_predict(X_train, trees, eta)))
            test_errors.append(mean_squared_error(y_test, gb_predict(X_test, trees, eta)))

        trees.append(tree)

    return trees, train_errors, test_errors
```

Теперь обучим несколько моделей с разными параметрами и исследуем их поведение.

Ввод [16]:
```
# Число деревьев в ансамбле
n_trees = 10

# Максимальная глубина деревьев
max_depth = 3

# Шаг
eta = 1

trees, train_errors, test_errors = gb_fit(n_trees, max_depth, X_train, X_test, y_train, y_test, eta)
```

Ввод [17]:
```
def evaluate_alg(X_train, X_test, y_train, y_test, trees, eta):
    train_prediction = gb_predict(X_train, trees, eta)

    print(f'Ошибка алгоритма из {n_trees} деревьев глубиной {max_depth} \
    с шагом {eta} на тренировочной выборке: {mean_squared_error(y_train, train_prediction)}')

    test_prediction = gb_predict(X_test, trees, eta)

    print(f'Ошибка алгоритма из {n_trees} деревьев глубиной {max_depth} \
    с шагом {eta} на тестовой выборке: {mean_squared_error(y_test, test_prediction)}')
```

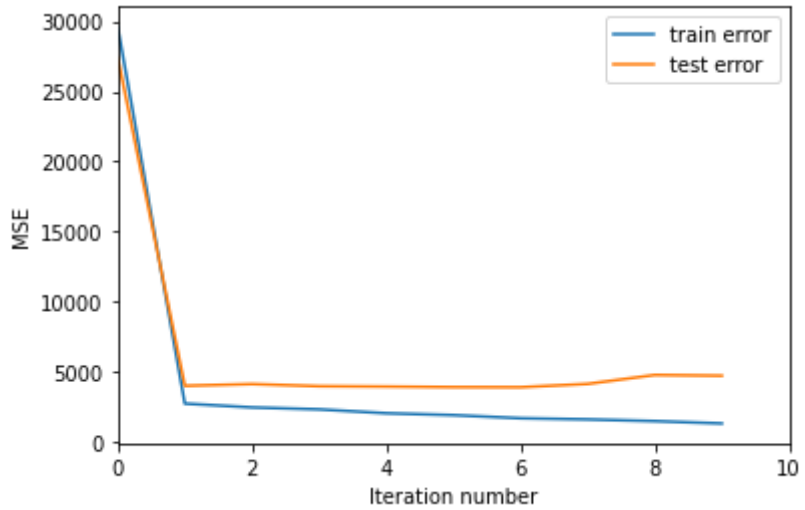
Ввод [18]:
```
evaluate_alg(X_train, X_test, y_train, y_test, trees, eta)
```

```
Ошибка алгоритма из 10 деревьев глубиной 3     с шагом 1 на тренировочной выборке: 1245.3540276511226
Ошибка алгоритма из 10 деревьев глубиной 3     с шагом 1 на тестовой выборке: 4818.854923599022
```

Построим графики зависимости ошибки на обучающей и тестовой выборках от числа итераций.

Ввод [19]:
```
def get_error_plot(n_trees, train_err, test_err):
    plt.xlabel('Iteration number')
    plt.ylabel('MSE')
    plt.xlim(0, n_trees)
    plt.plot(list(range(n_trees)), train_err, label='train error')
    plt.plot(list(range(n_trees)), test_err, label='test error')
    plt.legend(loc='upper right')
    plt.show()
```

Ввод [20]:
```
get_error_plot(n_trees, train_errors, test_errors)
```



Такой результат не является удовлетворительным

Увеличим число деревьев.

Ввод [21]:

```
n_trees = 50

trees, train_errors, test_errors = gb_fit(n_trees, max_depth, X_train, X_test, y_train, y_test, eta)
```

```
CPU times: user 1min 42s, sys: 600 ms, total: 1min 43s
Wall time: 1min 44s
```

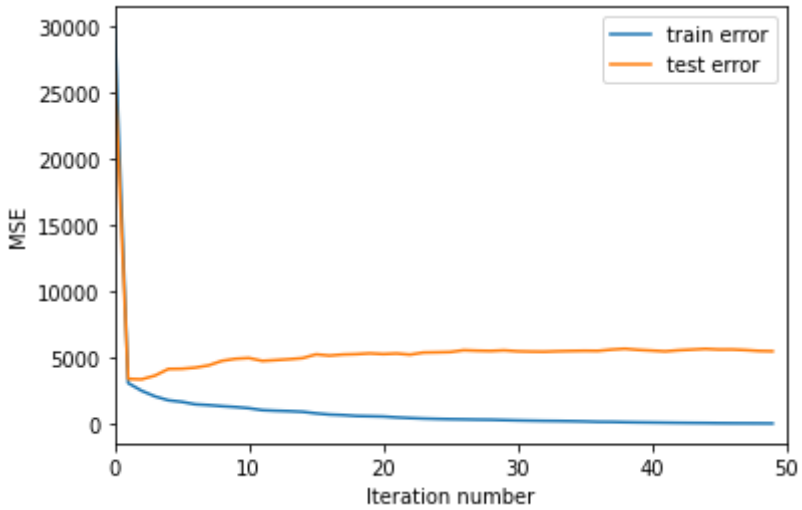
Ввод [60]:

```
evaluate_alg(X_train, X_test, y_train, y_test, trees, eta)
```

```
Ошибка алгоритма из 50 деревьев глубиной 3     с шагом 1 на тренировочной выборке: 70.43861643653375
Ошибка алгоритма из 50 деревьев глубиной 3     с шагом 1 на тестовой выборке: 5513.332418598973
```

Ввод [61]:

```
get_error_plot(n_trees, train_errors, test_errors)
```



Теперь попробуем уменьшить шаг.

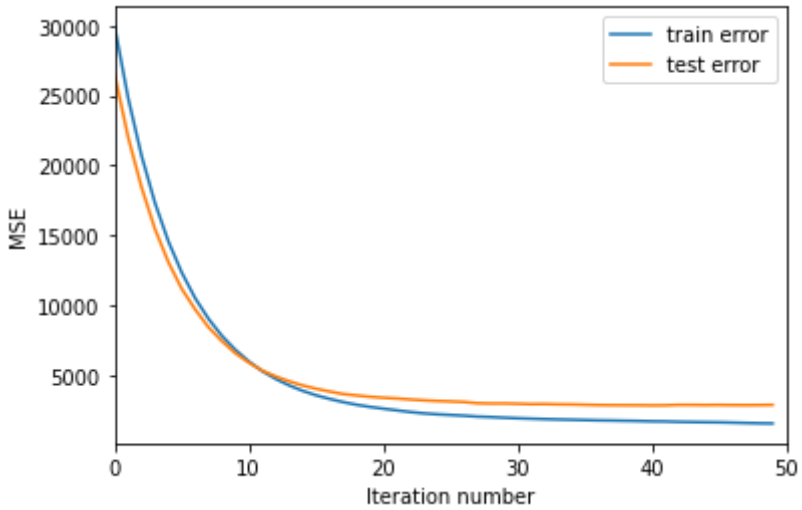
Ввод [62]:
```
%%time
eta = 0.1

trees, train_errors, test_errors = gb_fit(n_trees, max_depth, X_train, X_test, y_train, y_test, eta)
```

```
CPU times: user 3min 15s, sys: 152 ms, total: 3min 15s
Wall time: 3min 16s
```

Ввод [63]:
```
evaluate_alg(X_train, X_test, y_train, y_test, trees, eta)
```

```
Ошибка алгоритма из 50 деревьев глубиной 3     с шагом 0.1 на тренировочной выборке: 1522.5895529378424
Ошибка алгоритма из 50 деревьев глубиной 3     с шагом 0.1 на тестовой выборке: 2874.4849477323646
```

Ввод [64]:
```
get_error_plot(n_trees, train_errors, test_errors)
```



Видим, что качество обучения улучшается.

Уменьшим шаг до 0.01.

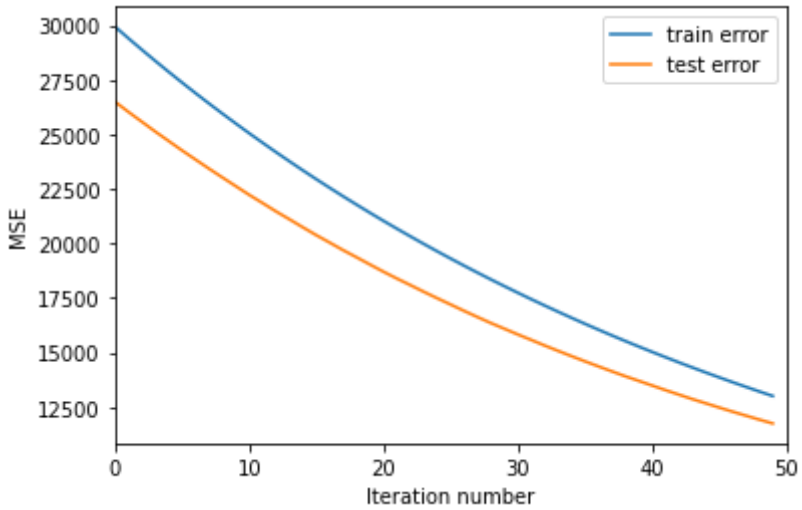
Ввод [65]:
```
%%time
eta = 0.01

trees, train_errors, test_errors = gb_fit(n_trees, max_depth, X_train, X_test, y_train, y_test, eta)
```

```
CPU times: user 3min 17s, sys: 164 ms, total: 3min 17s
Wall time: 3min 17s
```

Ввод [66]:
```
evaluate_alg(X_train, X_test, y_train, y_test, trees, eta)
```

```
Ошибка алгоритма из 50 деревьев глубиной 3     с шагом 0.01 на тренировочной выборке: 12801.091549463747
Ошибка алгоритма из 50 деревьев глубиной 3     с шагом 0.01 на тестовой выборке: 11578.276199882108
```

Ввод [67]:
```
get_error_plot(n_trees, train_errors, test_errors)
```



При таком размере шага алгоритм сходится, но ему для достижения удовлетворительных показателей требуется большее количество итераций.

Вернемся к шагу 0.1 и попробуем увеличить глубину деревьев

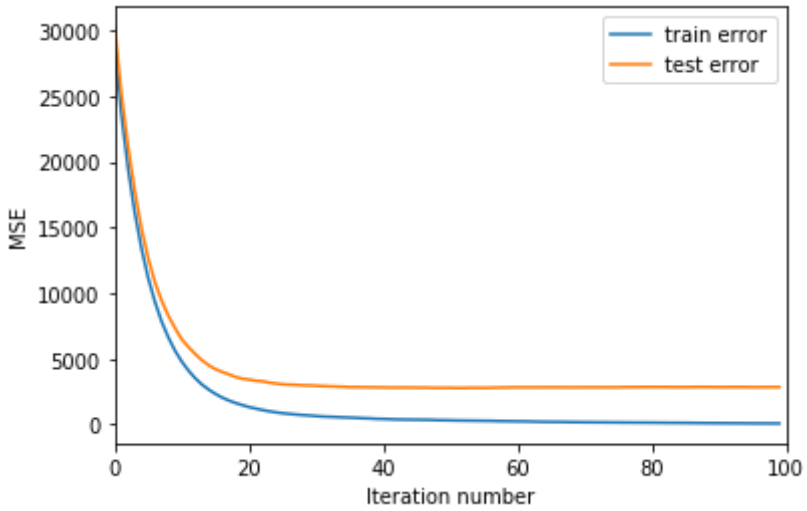
Ввод [44]:
```
%%time
eta = 0.1
max_depth = 5

trees, train_errors, test_errors = gb_fit(n_trees, max_depth, X_train, X_test, y_train, y_test, eta)
```

```
CPU times: user 7min 11s, sys: 256 ms, total: 7min 12s
Wall time: 7min 13s
```

Ввод [45]:
```
evaluate_alg(X_train, X_test, y_train, y_test, trees, eta)
```

```
Ошибка алгоритма из 100 деревьев глубиной 5     с шагом 0.1 на тренировочной выборке: 75.58799004984678
Ошибка алгоритма из 100 деревьев глубиной 5     с шагом 0.1 на тестовой выборке: 2848.374512516672
```

Ввод [46]:
```
get_error_plot(n_trees, train_errors, test_errors)
```



Ошибка на обучающей выборке упала, а на тестовой несколько поднялась, то есть в данном случае можем говорить о появлении переобучения.

В целом, тут мы показали, что варьируя параметры обучения градиентного бустинга можно добиваться различного уровня точности модели.

Существуют различные реализации градиентного бустинга, и одна из самых популярных и широко используемых - XGBoost (в Python содержится в библиотеке с аналогичным названием). С этой реализацией можно ознакомиться в дополнительных материалах.

AdaBoost

Видео (https://www.youtube.com/watch?v=LsK-xG1cLYA) с подробным объяснением алгоритма

Для задачи бинарной классификации он заключается в использовании слабых классификаторов (например, деревьев глубиной 1 - так называемых "пней") в цикле, с придаванием объектам весов. После каждого шага итерации, когда разделяющая плоскость классификатора делит пространство объектов на две части, веса объектов перераспределяются, и веса неправильно классифицированных объектов увеличиваются, чтобы на следующей итерации классификатор акцентировался на этих объектах. Классификатору также присваивается вес в зависимости от его точности. Затем полученные деревья с весами объединяются в один сильный классификатор. В этом и заключается адаптивность алгоритма. Алгоритм AdaBoost также называют алгоритмом усиления классификаторов.

Продемонстрируем работу AdaBoost

Ввод [21]:
```
from sklearn.tree import DecisionTreeClassifier, plot_tree
from sklearn.datasets import load_breast_cancer
```

Ввод [68]:
```
X, y = load_breast_cancer(return_X_y=True, as_frame=True)
X.shape, y.shape
```

Out[68]:
```
((569, 30), (569,))
```

Ввод [69]:
```
np.random.seed(6)
index = np.random.randint(0, X.shape[0], 1)
X_test = X.loc[index]
y_test = y.loc[index]
```

Ввод [24]:
```
X.columns
```

Out[24]:
```
Index(['mean radius', 'mean texture', 'mean perimeter', 'mean area',
       'mean smoothness', 'mean compactness', 'mean concavity',
       'mean concave points', 'mean symmetry', 'mean fractal dimension',
       'radius error', 'texture error', 'perimeter error', 'area error',
       'smoothness error', 'compactness error', 'concavity error',
       'concave points error', 'symmetry error', 'fractal dimension error',
       'worst radius', 'worst texture', 'worst perimeter', 'worst area',
       'worst smoothness', 'worst compactness', 'worst concavity',
       'worst concave points', 'worst symmetry', 'worst fractal dimension'],
      dtype='object')
```

Ввод [70]:
```
X = X.loc[[ 41,  44,  73,  81,  89,  91, 135, 146, 484, 491]]
y = y.loc[[ 41,  44,  73,  81,  89,  91, 135, 146, 484, 491]]
X
```

Out[70]:

	mean radius	mean texture	mean perimeter	mean area	mean smoothness	mean compactness	mean concavity	mean concave points	mean symmetry	mean fractal dimension	...	worst radius	worst texture	worst perimeter	wors area
41	10.95	21.35	71.90	371.1	0.12270	0.12180	0.10440	0.05669	0.1895	0.06870	...	12.84	35.34	87.22	514.(
44	13.17	21.81	85.42	531.5	0.09714	0.10470	0.08259	0.05252	0.1746	0.06177	...	16.23	29.89	105.50	740.7
73	13.80	15.79	90.43	584.1	0.10070	0.12800	0.07789	0.05069	0.1662	0.06566	...	16.57	20.86	110.30	812.4
81	13.34	15.86	86.49	520.0	0.10780	0.15350	0.11690	0.06987	0.1942	0.06902	...	15.53	23.19	96.66	614.9
89	14.64	15.24	95.77	651.9	0.11320	0.13390	0.09966	0.07064	0.2116	0.06346	...	16.34	18.24	109.40	803.(
91	15.37	22.76	100.20	728.2	0.09200	0.10360	0.11220	0.07483	0.1717	0.06097	...	16.43	25.84	107.50	830.9
135	12.77	22.47	81.72	506.3	0.09055	0.05761	0.04711	0.02704	0.1585	0.06065	...	14.49	33.37	92.04	653.(
146	11.80	16.58	78.99	432.0	0.10910	0.17000	0.16590	0.07415	0.2678	0.07371	...	13.74	26.38	91.93	591.7
484	15.73	11.28	102.80	747.2	0.10430	0.12990	0.11910	0.06211	0.1784	0.06259	...	17.01	14.20	112.50	854.3
491	17.85	13.23	114.60	992.1	0.07838	0.06217	0.04445	0.04178	0.1220	0.05243	...	19.82	18.42	127.10	1210.(

10 rows × 30 columns

Ввод [26]:
```
y
```

Out[26]:
```
41     0
44     0
73     0
81     1
89     1
91     0
135    0
146    0
484    1
491    1
Name: target, dtype: int64
```

1. Инициализация начальных весов объектов из выборки длиной l:

$$D_1(i) = \frac{1}{l}$$

Ввод [27]:
```
n_objects = X.shape[0]
w = np.ones(n_objects) / n_objects
w
```

Out[27]:
```
array([0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1])
```

2. Для каждого из N деревьев в ансамбле:

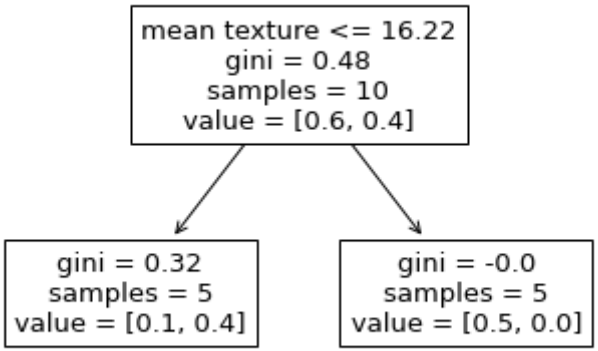
- находим классификатор b_n, который минимизирует взвешенную ошибку классификации

$$b_n = \underset{b}{\operatorname{argmin}}\ \varepsilon_j,$$

где

$$\varepsilon_j = \sum_{i=1}^{l} D_n(i)[y_i \neq b_j(x)]$$

Ввод [28]:
```
stump1 = DecisionTreeClassifier(max_depth=1, random_state=1)
stump1.fit(X, y, sample_weight=w)

pred = stump1.predict(X)
plot_tree(stump1, feature_names=X.columns, );
```



Ввод [29]:
```
pred == y
```

Out[29]:
```
41      True
44      True
73     False
81      True
89      True
91      True
135     True
146     True
484     True
491     True
Name: target, dtype: bool
```

Ввод [30]:
```
error1 = sum(pred != y) / len(y)
error1
```

Out[30]:
```
0.1
```

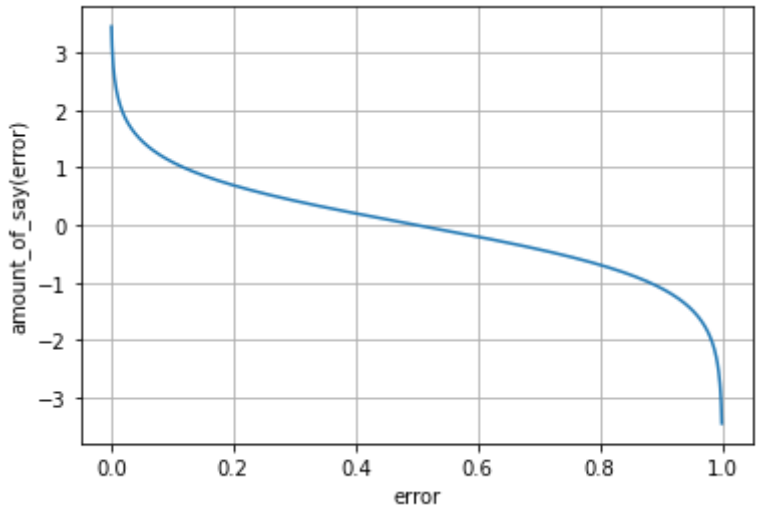
- критерием остановки является значение $\varepsilon_j \geq 0.5$. При таком значении ошибки нужно выбрать другой классификатор и продолжить.
- выбираем вес для дерева α_n по формуле

$$\alpha_n = \frac{1}{2}\ln\frac{1-\varepsilon_n}{\varepsilon_n}$$

Ввод [31]:
```
def amount_of_say(x):
    return 1/2 * np.log((1 - x) / x)
```

Ввод [32]:
```
dots = np.linspace(0, 1, 1000)
amount_of_say_value = list(map(amount_of_say, dots))

plt.xlabel('error')
plt.ylabel('amount_of_say(error)')
plt.grid()
plt.plot(dots, amount_of_say_value);
```

```
<ipython-input-31-d4f702aa4b94>:2: RuntimeWarning: divide by zero encountered in double_scalars
  return 1/2 * np.log((1 - x) / x)
<ipython-input-31-d4f702aa4b94>:2: RuntimeWarning: divide by zero encountered in log
  return 1/2 * np.log((1 - x) / x)
```



Получим вес для пня

Ввод [33]:
```
alpha1 = 1/2 * np.log((1 - error1) / error1)
alpha1
```

Out[33]:
```
1.0986122886681098
```

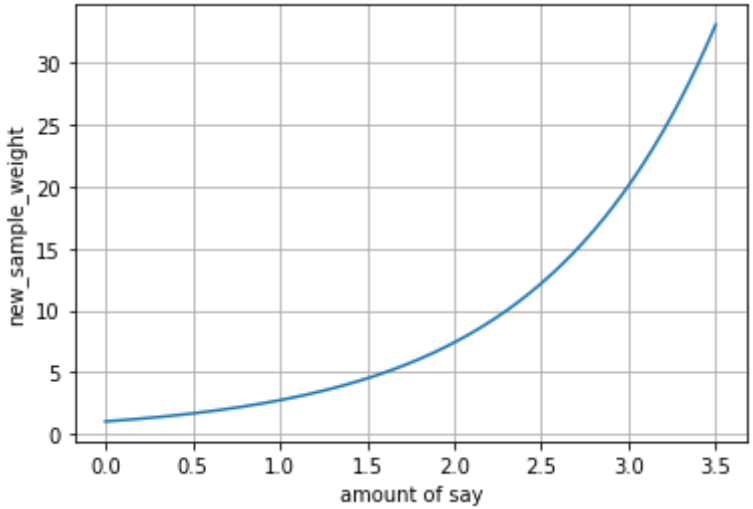
- обновляем веса при объектах:

Ввод [34]:
```
def new_sample_weight(x):
    return np.exp(x)
```

Ввод [72]:
```
0.1 * np.exp(3.5)
```

Out[72]:
```
3.3115451958692312
```

Ввод [36]:
```
dots = np.linspace(0, 3.5, 1000)
new_sample_weight_value = list(map(new_sample_weight, dots))

plt.xlabel('amount of say')
plt.ylabel('new_sample_weight')
plt.grid()
plt.plot(dots, new_sample_weight_value);
```



Если пень сделал не очень хорошую классификацию, то вес объекта станет немного больше, если пень сделал хорошую классификацию, то вес объекта станет больше.

Меняем вес неверное классифицированного объекта

Ввод [37]:
```
wrong_mask = pred != y
w[wrong_mask] = w[wrong_mask] * np.exp(alpha1)
w
```

Out[37]:
```
array([0.1, 0.1, 0.3, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1])
```

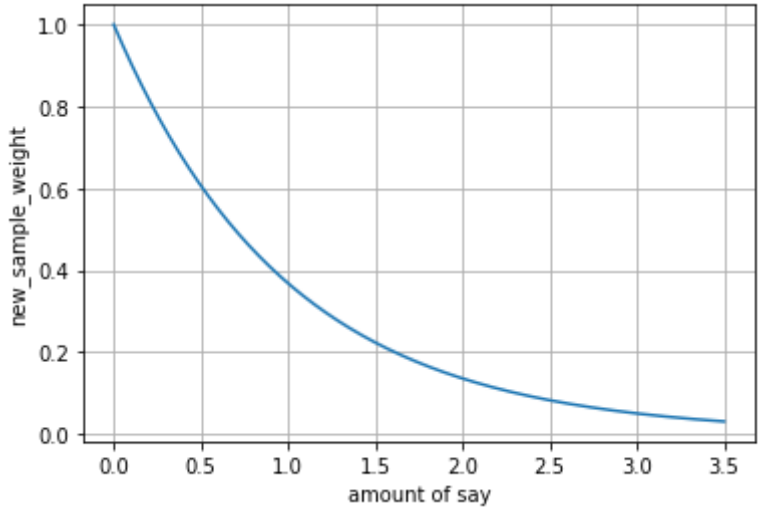
$D_i = D_{i-1} e^{-\alpha_i}$ – изменение веса одного объекта при верной классификации

Ввод [38]:
```
def new_sample_weight(x):
    return np.exp(-x)
```

Ввод [73]:
```
0.1 * np.exp(-3.5)
```

Out[73]:
```
0.00301973834223185
```

Ввод [40]:
```
dots = np.linspace(0, 3.5, 1000)
new_sample_weight_value = list(map(new_sample_weight, dots))

plt.xlabel('amount of say')
plt.ylabel('new_sample_weight')
plt.grid()
plt.plot(dots, new_sample_weight_value);
```



Если пень сделал не очень хорошую классификацию, то вес объекта станет меньше, если пень сделал хорошую классификацию, то вес объекта станет немного меньше.

То есть, те объекты, которые хорошо классифицируются будут иметь меньший вес, чем те, на которых классификатор ошибается.

Меняем веса верно классифицированных объектов

Ввод [41]:
```
w[~wrong_mask] = w[~wrong_mask] * np.exp(-alpha1)
w
```

Out[41]:
```
array([0.03333333, 0.03333333, 0.3       , 0.03333333, 0.03333333,
       0.03333333, 0.03333333, 0.03333333, 0.03333333, 0.03333333])
```

Ввод [42]:
```
sum(w)
```

Out[42]:
```
0.6
```

Ввод [43]:
```
w /= sum(w)
```

Ввод [44]:
```
w
```

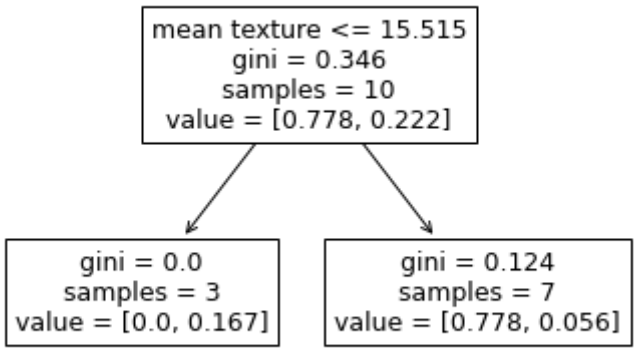
Out[44]:
```
array([0.05555556, 0.05555556, 0.5       , 0.05555556, 0.05555556,
       0.05555556, 0.05555556, 0.05555556, 0.05555556, 0.05555556])
```

Ввод [45]:
```
sum(w)
```

Out[45]:
```
1.0000000000000002
```

Обучим следующий пень

Ввод [46]:
```
stump2 = DecisionTreeClassifier(max_depth=1, random_state=1)
stump2.fit(X, y, sample_weight=w)

pred = stump2.predict(X)
plot_tree(stump2, feature_names=X.columns);
```



Ввод [47]:
```
pred != y
```

Out[47]:
```
41     False
44     False
73     False
81      True
89     False
91     False
135    False
146    False
484    False
491    False
Name: target, dtype: bool
```

Ввод [48]:
```
error2 = sum(pred != y) / len(y)
alpha2 = 1/2 * np.log((1 - error2) / error2)

wrong_mask = pred != y
w[wrong_mask] = w[wrong_mask] * np.exp(alpha2)

w[~wrong_mask] = w[~wrong_mask] * np.exp(-alpha2)
w /= sum(w)
w
```

Out[48]:
```
array([0.03846154, 0.03846154, 0.34615385, 0.34615385, 0.03846154,
       0.03846154, 0.03846154, 0.03846154, 0.03846154, 0.03846154])
```

Обучим следующий пень

Ввод [49]:
```
stump3 = DecisionTreeClassifier(max_depth=1, random_state=1)
stump3.fit(X, y, sample_weight=w)

pred = stump3.predict(X)
error3 = sum(pred != y) / len(y)
alpha3 = 1/2 * np.log((1 - error3) / error3)

wrong_mask = pred != y
w[wrong_mask] = w[wrong_mask] * np.exp(alpha3)

w[~wrong_mask] = w[~wrong_mask] * np.exp(-alpha3)
w /= sum(w)
w
```

Out[49]:
```
array([0.03125, 0.03125, 0.28125, 0.28125, 0.125  , 0.03125, 0.03125,
       0.03125, 0.03125, 0.125  ])
```

Ввод [50]:
```
wrong_mask
```

Out[50]:
```
41     False
44     False
73     False
81     False
89      True
91     False
135    False
146    False
484    False
491     True
Name: target, dtype: bool
```

Предскажем классы объектов с помощью трех пней

Ввод [51]:
```
display(X_test, y_test)
```

	mean radius	mean texture	mean perimeter	mean area	mean smoothness	mean compactness	mean concavity	mean concave points	mean symmetry	mean fractal dimension	...	worst radius	worst texture	worst perimeter	worst area
227	15.0	15.51	97.45	684.5	0.08371	0.1096	0.06505	0.0378	0.1881	0.05907	...	16.41	19.31	114.2	808.2

1 rows × 30 columns

```
227    1
Name: target, dtype: int64
```

Ввод [52]:
```
alpha1, alpha2, alpha3
```

Out[52]:
```
(1.0986122886681098, 1.0986122886681098, 0.6931471805599453)
```

Ввод [53]:
```
pred1 = stump1.predict(X_test)
pred2 = stump2.predict(X_test)
pred3 = stump3.predict(X_test)
pred1, pred2, pred3
```

Out[53]:
```
(array([1]), array([1]), array([0]))
```

Ввод [54]:
```
alpha1 + alpha2, alpha3
```

Out[54]:
```
(2.1972245773362196, 0.6931471805599453)
```

Ввод [55]:
```
pred = 1
y_test
```

Out[55]:
```
227    1
Name: target, dtype: int64
```

Алгоритм AdaBoost

1. Инициализация начальных весов объектов из выборки длиной l (равномерно):

$$D_1(i) = \frac{1}{l}$$

2. Для каждого из N деревьев в ансамбле:

- находим классификатор b_n, который минимизирует взвешенную ошибку классификации

$$b_n = \underset{b}{\operatorname{argmin}}\ \varepsilon_j,$$

где

$$\varepsilon_j = \sum_{i=1}^{l} D_n(i)[y_i \neq b_j(x)]$$

$D_n(i)$ - вес объекта, $[y_i \neq b_j(x)]$ - неправильно классифицированные объекты

- критерием остановки является значение $\varepsilon_j \geq 0.5$. При таком значении ошибки нужно выбрать другой классификатор и продолжить.
- выбираем вес для дерева α_n по формуле

$$\alpha_n = \frac{1}{2}\ln\frac{1-\varepsilon_n}{\varepsilon_n}$$

- обновляем веса при объектах:

$$D_{n+1}(i) = \frac{D_n(i)e^{-\alpha_n y_i b_n(x_i)}}{Z_n},$$

выражение $y_i b_n(x_i)$ в случае $Y = \{-1, 1\}$ будет равняться 1 для правильно классифицированных объектов и -1 для неправильно классифицированных, то есть по сути правильность классификации будет означать, будет e^{α_n} стоять в числителе (увеличивается вес неправильно классиф. объектов) или в знаменателе (уменьшается вес правильно классиф. объектов) формулы. В случае $Y = \{0, 1\}$ вес будет уменьшаться у правильно классифицированных объектов, а у неправильно классифицированных - оставаться неизменным (до нормализации). Z_n здесь - нормализующий параметр, выбираемый так, чтобы D_{n+1} по своей сути являлся распределением вероятностей, то есть

$$\sum_{i=1}^{l} D_{n+1} = 1.$$

Реализация алгоритма AdaBoost

Ввод [56]:
```
from sklearn.tree import DecisionTreeClassifier
from sklearn.datasets import load_breast_cancer
```

Ввод [57]:
```
X, y = load_breast_cancer(return_X_y=True)
X.shape, y.shape
```

Out[57]:
```
((569, 30), (569,))
```

Разделим выборку на обучающую и тестовую

Ввод [58]:
```
X_train, X_test, y_train, y_test = model_selection.train_test_split(X, y, test_size=0.25, random_state=32)
```

Реализуем функцию подсчета ошибки

Ввод [59]:
```
def get_error(pred, y):
    return sum(pred != y) / len(y)
```

И сам алгоритм

Ввод [60]:

```
def adaboost(X, y, N):

    # Размер выборки
    n_objects = len(X)

    # Запишем количество классов в переменную
    n_classes = len(np.unique((y)))

    # Начальные веса объектов
    w = np.ones(n_objects) / n_objects

    # Деревья с весами будем записывать в список
    models = []

    for n in range(N):
        # Зададим дерево и обучим его
        clf = DecisionTreeClassifier(max_depth=1)
        clf.fit(X, y, sample_weight=w)

        predictions = clf.predict(X)
        error = get_error(predictions, y)

        # отбросим дерево, если его ошибка больше 0.5
        # Запишем условие в общем виде (применимо к небинарным классификаторам)
        if error >= 1 - 1/n_classes:
            continue

        # Обработаем граничные значения ошибок
        if error == 0:
            error += 1e-10

        # Вычислим вес для дерева
        alpha = 0.5 * np.log((1 - error) / error)


        # Найдем индексы правильно классифицированных элементов
        wrong_mask = predictions != y

        # Увеличим веса для неправильно классифицированных элементов
        w[wrong_mask] *= np.exp(alpha)
        # Уменьшаем веса для правильно классифицированных элементов
```

```
        w[~wrong_mask] *= np.exp(-alpha)

        # Нормализуем веса
        w /= w.sum()

        # Добавим дерево с весом в список
        models.append((alpha, clf))

    return models
```

Обучим алгоритм из 50 деревьев

Ввод [61]:
```
N = 50

models = adaboost(X_train, y_train, N)
```

Теперь осуществим предсказание

Ввод [62]:
```
np.zeros((10, 2))
```

Out[62]:
```
array([[0., 0.],
       [0., 0.],
       [0., 0.],
       [0., 0.],
       [0., 0.],
       [0., 0.],
       [0., 0.],
       [0., 0.],
       [0., 0.],
       [0., 0.]])
```

Ввод [63]:
```
def predict(X, models):

    n_classes = 2
    n_objects = len(X)

    # вначале обозначим предсказание нулевым массивом
    y_pred = np.zeros((n_objects, n_classes))

    for alpha, clf in models:
        prediction = clf.predict(X)
        # Для каждого предсказания будем прибавлять alpha к
        # элементу с индексом предсказанного класса
        y_pred[range(n_objects), prediction] += alpha

    # выберем индексы с максимальными суммарными весами -
    # получим предсказанные алгоритмом классы
    y_pred = np.argmax(y_pred, axis=1)

    return y_pred
```

Ввод [64]:
```
print(f'Точность алгоритма на обучающей выборке: {(1 - get_error(predict(X_train, models), y_train)) * 100:.3f}')
```

```
Точность алгоритма на обучающей выборке: 96.948
```

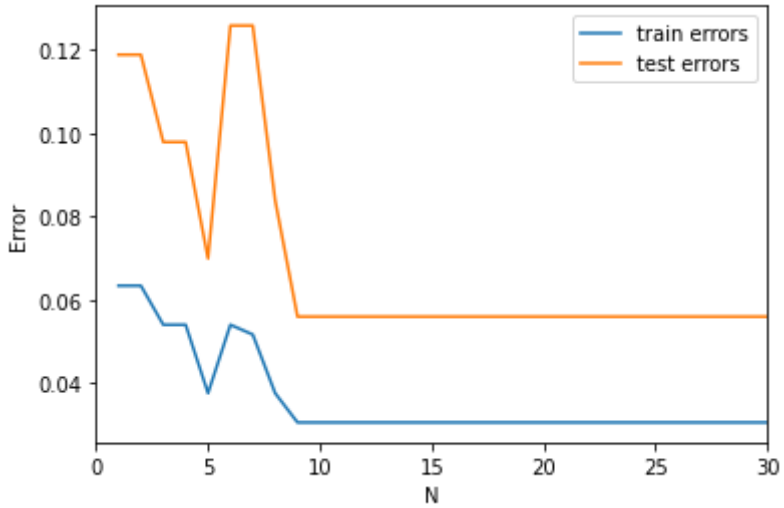
Ввод [65]:
```
print(f'Точность алгоритма на тестовой выборке: {(1 - get_error(predict(X_test, models), y_test)) * 100:.3f}')
```

```
Точность алгоритма на тестовой выборке: 94.406
```

Построим графики зависимости ошибки от количества базовых алгоритмов в ансамбле.

Ввод [66]:
```
train_errors = []
test_errors = []

for n in range(1, 31):
    models = adaboost(X_train, y_train, n)
    train_errors.append(get_error(predict(X_train, models), y_train))
    test_errors.append(get_error(predict(X_test, models), y_test))
```

Ввод [67]:
```
x = list(range(1, 31))

plt.xlim(0, 30)
plt.plot(x, train_errors, label='train errors')
plt.plot(x, test_errors, label='test errors')
plt.xlabel('N')
plt.ylabel('Error')
plt.legend(loc='upper right');
```



Достоинствами алгоритма AdaBoost можно назвать простоту реализации, хорошую обобщающую способность и небольшую вычислительную сложность. В то же время, есть и недостатки - в первую очередь, склонность к переобучению при наличии в данных шума и выбросов: для наиболее труднoклассифицируемых объектов алгоритм будет определять очень большие веса и в итоге переобучаться на них. В то же время это является и плюсом: таким образом можно идентифицировать выбросы.

Домашнее задание

1. Для реализованной модели градиентного бустинга построить графики зависимости ошибки от количества деревьев в ансамбле и от максимальной глубины деревьев. Сделать выводы о зависимости ошибки от этих параметров.
2. *Модифицировать реализованный алгоритм градиентного бустинга, чтобы получился стохастический градиентный бустинг. Размер подвыборки принять равным 0.5. Сравнить на одном графике кривые изменения ошибки на тестовой выборке в зависимости от числа итераций.
3. *Оптимизировать процесс обучения градиентного бустинга, чтобы он занимал меньше времени.

Проект*:

1. https://www.kaggle.com/c/gb-tutors-expected-math-exam-results (https://www.kaggle.com/c/gb-tutors-expected-math-exam-results) регрессия
2. https://www.kaggle.com/c/gb-choose-tutors (https://www.kaggle.com/c/gb-choose-tutors) классификация

Дополнительные материалы

1. Интерактивная демонстрация градиентного бустинга (http://arogozhnikov.github.io/2016/06/24/gradient_boosting_explained.html)
2. sklearn.datasets (https://scikit-learn.org/stable/datasets/index.html)
3. sklearn.tree.DecisionTreeRegressor (https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeRegressor.html)
4. L_1 loss и L_2 loss (https://afteracademy.com/blog/what-are-l1-and-l2-loss-functions)
5. XGBoost (https://github.com/esokolov/ml-course-hse/blob/master/2016-fall/lecture-notes/lecture10-ensembles.pdf)
6. AdaBoost (https://ru.wikipedia.org/wiki/AdaBoost)
7. XGBoost: A Scalable Tree Boosting System - оригинальная статья (http://scholar.google.ru/scholar_url?url=https://dl.acm.org/ft_gateway.cfm%3Fftid%3D1775849%26id%3D2939785&hl=en&sa=X&scisig=AAGBfm3b8fqJWtjjjejQ5fQwrtg9eQQK-w&nossl=1&oi=scholarr)

Summary

- На больших и сложных данных градиентный бустинг - один из лучших алгоритмов
- Много настраиваемых параметров
- Есть очень быстрые реализации
- Обычно строят на деревьях решений

Определения

Бустинг

Бустинг — это техника построения ансамблей, в которой предсказатели построены не независимо, а последовательно.

Алгоритм построения градиентного бустинга

1. Инициализация начального алгоритма $b_0(x)$
2. Цикл по n = 1,2,3,...:

   - Подсчитывание остатков $s = \left(-\frac{\partial L}{\partial z}\Big|_{z=a_{n-1}(x_1)}, \ldots, -\frac{\partial L}{\partial z}\Big|_{z=a_{n-1}(x_l)} \right);$
   - Обучение нового алгоритма $b_n(x) = \underset{s}{\operatorname{argmin}} \frac{1}{l} \sum_{i=1}^{l} (b(x_i) - s_i)^2;$
   - Добавление алгоритма в композицию $a_n(x) = a_{n-1}(x) + \eta b_n(x).$

Алгоритм построения AdaBoost

1. Инициализация начальных весов объектов: $D_1(i) = \frac{1}{l}$
2. Цикл по n = 1,2,3,...:

   - находим классификатор b_n, который минимизирует взвешенную ошибку классификации $b_n = \underset{b}{\operatorname{argmin}}\ \varepsilon_j$, где $\varepsilon_j = \sum_{i=1}^{l} D_n(i)[y_i \neq b_j(x)]$
   - выбираем вес для дерева α_n по формуле $\alpha_n = \frac{1}{2} \ln \frac{1-\varepsilon_n}{\varepsilon_n}$

- обновляем веса при объектах: $D_{n+1}(i) = \frac{D_n(i)e^{-\alpha_n y_i b_n(x_i)}}{Z_n}$,
- критерием останова является значение $\varepsilon_j \geq 0.5$. При таком значении ошибки нужно выбрать другой классификатор и продолжить.