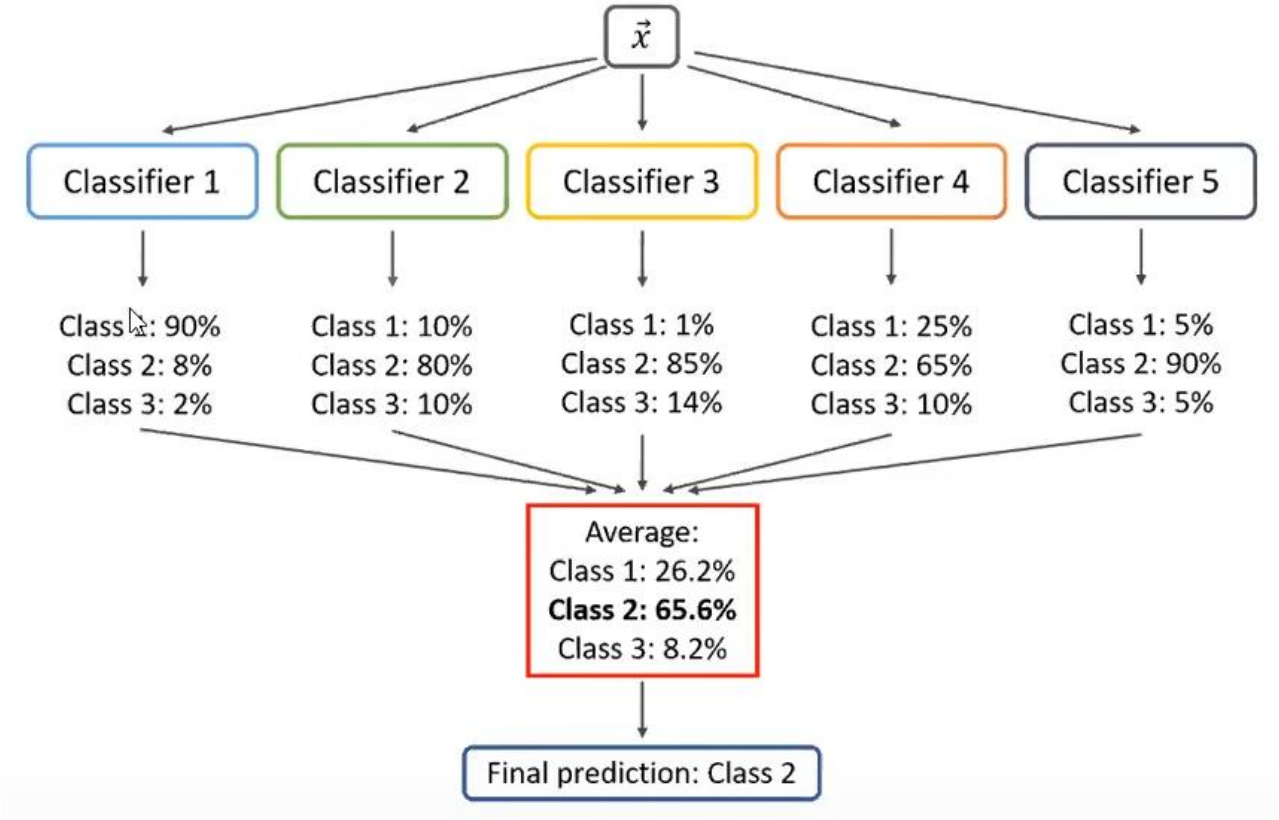
Ансамбли моделей

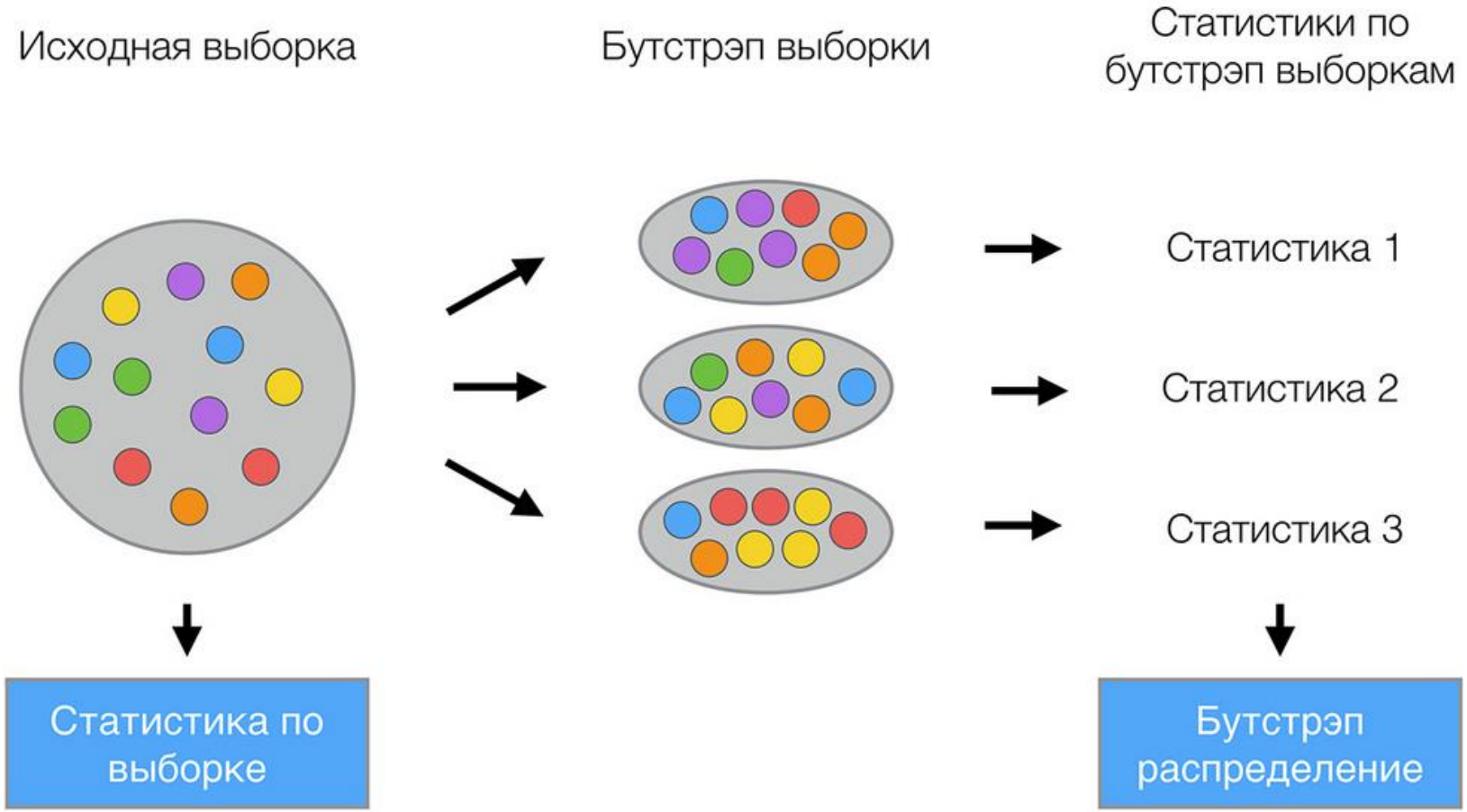
Ансамбль методов или моделей в ML использует несколько алгоритмов для увеличения показателей эффективности прогнозирования, чем при использовании одного метода.



- Сильно отличающиеся методы улучшают результаты,
- «Слабые» методы будут делать ансамбль слабее, могут «усреднить» результаты сильных методов.

Бутстрэп

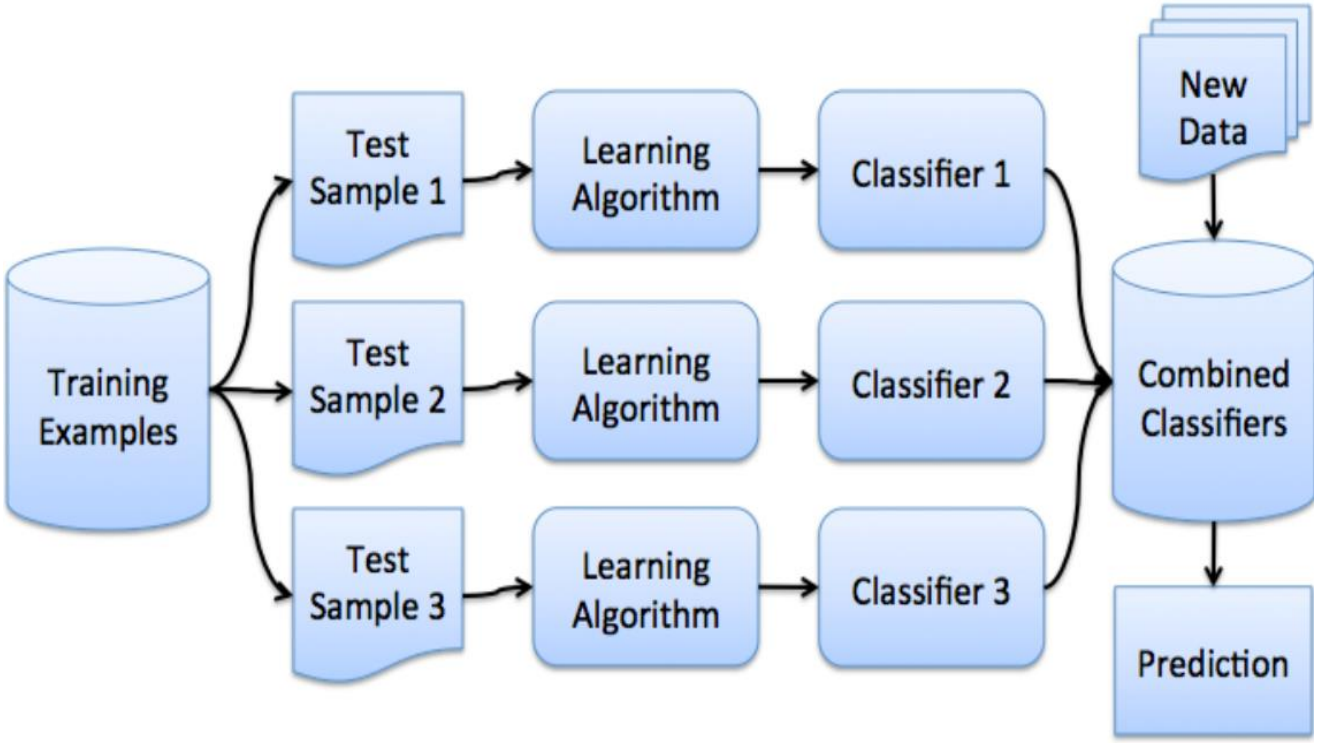
Выборка X размера N. Равномерно с возвращением возьмем из выборки N объектов (среди них могут быть повторы). Обозначим новую выборку через X_1. Повторяя процедуру M раз, сгенерируем M подвыборок X_1, …, X_M. Теперь можно оценивать различные статистики исходного распределения.



Bagging

Bagging == **B**ootstrap aggregation.

Пусть имеется обучающая выборка X. С помощью бутстрэпа сгенерируем из неё выборки X_1, …, X_M. На каждой выборке обучим свой классификатор $a_i(x)$. Итоговый классификатор будет усреднять ответы всех этих алгоритмов (в случае классификации это соответствует голосованию): $a(x) = \frac{1}{M}\sum_{i=1}^{M} a_i(x)$



Решающие деревья

Решающие деревья воспроизводят логические схемы для получения окончательного решения о классификации объекта с помощью ответов на иерархически организованную систему вопросов.
Дерево включает в себя корневую вершину (корень), «листья» и «ветки». На рёбрах («ветках») записаны признаки, от которых зависит целевая функция, в «листьях» записаны значения целевой функции, а в остальных узлах — признаки, по которым происходит разделение.

Случайный лес

Алгоритм построения **случайного леса**, состоящего из N деревьев:
Для каждого $n=1,\ldots,N$:

- Сгенерировать выборку X_n с помощью бутстрэпа;
- Построить решающее дерево b_n по выборке X_n :
  — по заданному критерию выбираем лучший признак, делаем разбиение в дереве по нему и так повторяем до исчерпания выборки
  — дерево строится, пока в каждом листе не более n_{min} объектов или пока не достигнем определенной высоты дерева
  — при каждом разбиении сначала выбирается m случайных признаков из n исходных, и оптимальное разделение выборки ищется только среди них.

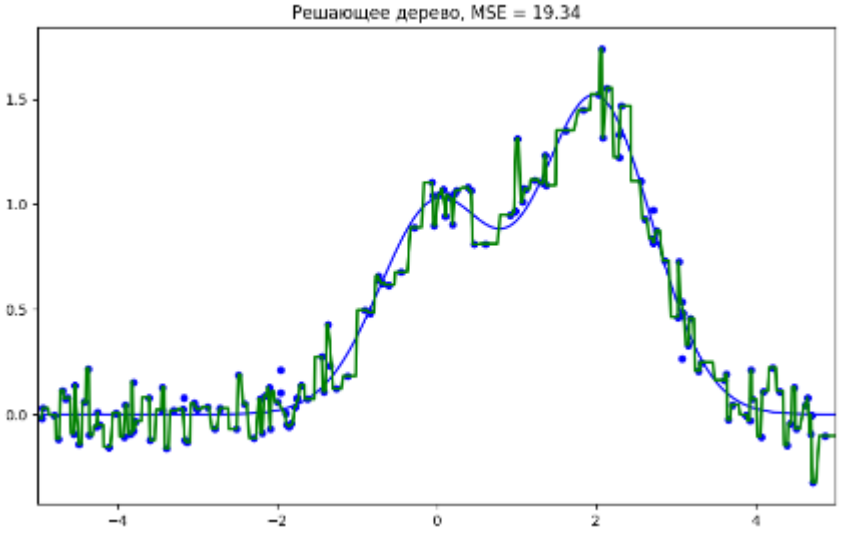
Итоговый классификатор $$a(x) = \frac{1}{n}\sum_{i=1}^{n} b_i(x)$$, при этом для задачи классификации выбираем решение голосованием по большинству, а в задаче регрессии — средним.

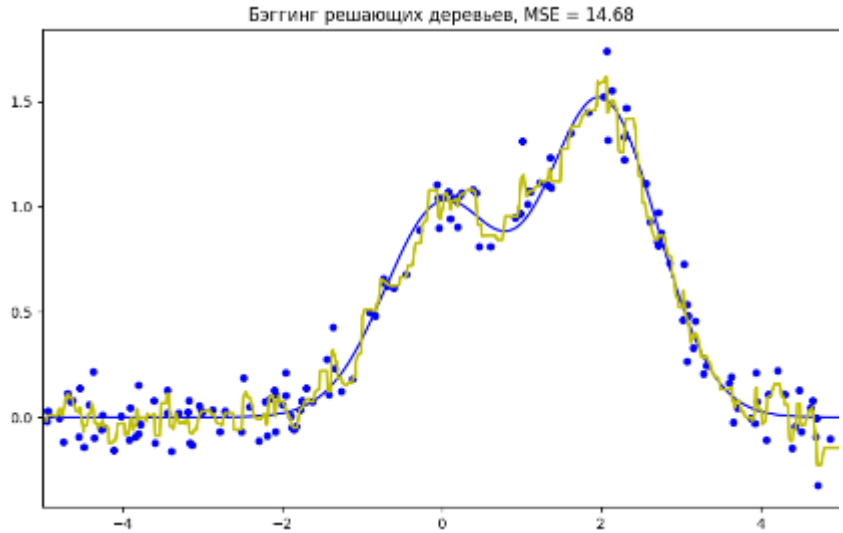
Рекомендуется в задачах классификации: $m = \sqrt{n}$, а в задачах регрессии — $m=n/3$, где n — количество признаков.
Таким образом, **случайный лес** — это бэггинг над решающими деревьями, при обучении которых для каждого разбиения признаки выбираются из некоторого случайного подмножества признаков.

Сравнения решающего дерева, бэггинга и случайного леса в задаче регрессии

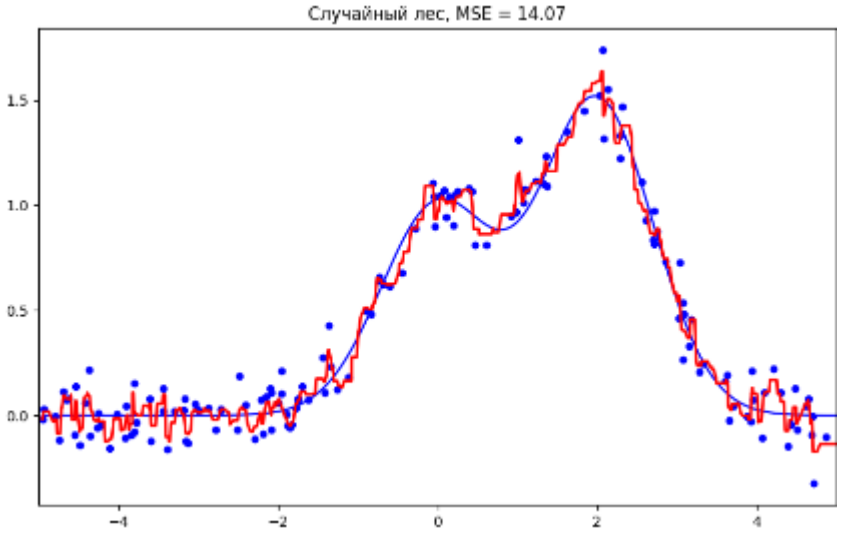
```
import matplotlib.pyplot as plt
import numpy as np; np.random.seed(42)
from sklearn.ensemble import RandomForestRegressor, RandomForestClassifier, BaggingRegressor
from sklearn.tree import DecisionTreeRegressor, DecisionTreeClassifier

n_train = 150; n_test = 1000; noise = 0.1

# Generate data
def f(x):
    x = x.ravel()
    return np.exp(-x ** 2) + 1.5 * np.exp(-(x - 2) ** 2)

def generate(n_samples, noise):
    X = np.random.rand(n_samples) * 10 - 5
    X = np.sort(X).ravel()
    y = np.exp(-X ** 2) + 1.5 * np.exp(-(X - 2) ** 2)\
        + np.random.normal(0.0, noise, n_samples)
    X = X.reshape((n_samples, 1))
    return X, y

X_train, y_train = generate(n_samples=n_train, noise=noise)
X_test, y_test = generate(n_samples=n_test, noise=noise)

# One decision tree regressor
dtree = DecisionTreeRegressor().fit(X_train, y_train)
d_predict = dtree.predict(X_test)

plt.figure(figsize=(10, 6))
plt.plot(X_test, f(X_test), "b")
plt.scatter(X_train, y_train, c="b", s=20)
plt.plot(X_test, d_predict, "g", lw=2)
plt.xlim([-5, 5])
plt.title("Решающее дерево, MSE = %.2f" % np.sum((y_test - d_predict) ** 2))

# Bagging decision tree regressor
bdt = BaggingRegressor(DecisionTreeRegressor()).fit(X_train, y_train)
bdt_predict = bdt.predict(X_test)

plt.figure(figsize=(10, 6))
plt.plot(X_test, f(X_test), "b")
plt.scatter(X_train, y_train, c="b", s=20)
plt.plot(X_test, bdt_predict, "y", lw=2)
plt.xlim([-5, 5])
plt.title("Бэггинг решающих деревьев, MSE = %.2f" % np.sum((y_test - bdt_predict) ** 2))

# Random Forest
rf = RandomForestRegressor(n_estimators=10).fit(X_train, y_train)
rf_predict = rf.predict(X_test)

plt.figure(figsize=(10, 6))
plt.plot(X_test, f(X_test), "b")
plt.scatter(X_train, y_train, c="b", s=20)
plt.plot(X_test, rf_predict, "r", lw=2)
plt.xlim([-5, 5])
plt.title("Случайный лес, MSE = %.2f" % np.sum((y_test - rf_predict) ** 2))
plt.show()
```

Сравнения решающего дерева, бэггинга и случайного леса в задаче регрессии





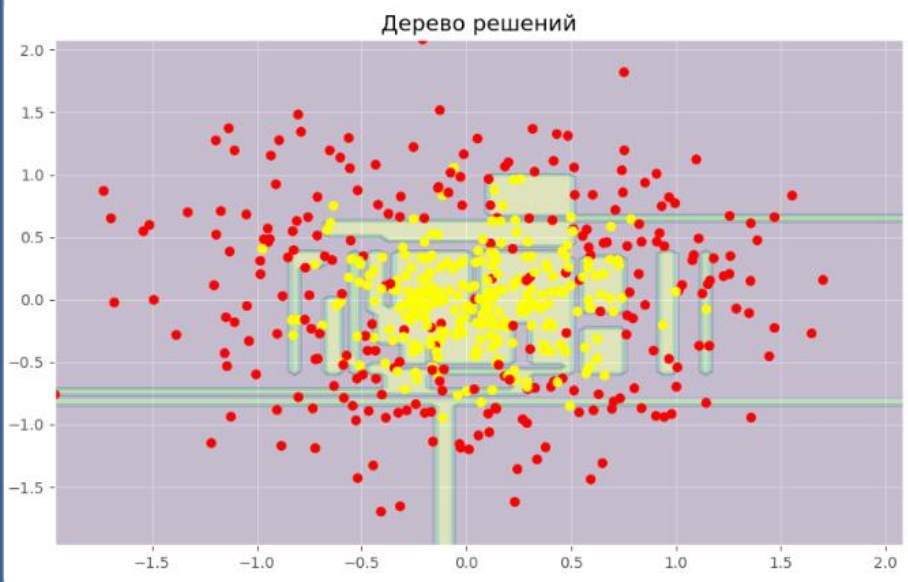
Случайный лес (Random Forest RF) из 10 деревьев дает лучший результат, чем одно дерево или бэггинг из 10 деревьев решений. Основное различие : в RF выбирается случайное подмножество признаков, лучший признак для разделения узла определяется из подвыборки признаков, в бэггинге – все функции рассматриваются для разделения в узле.

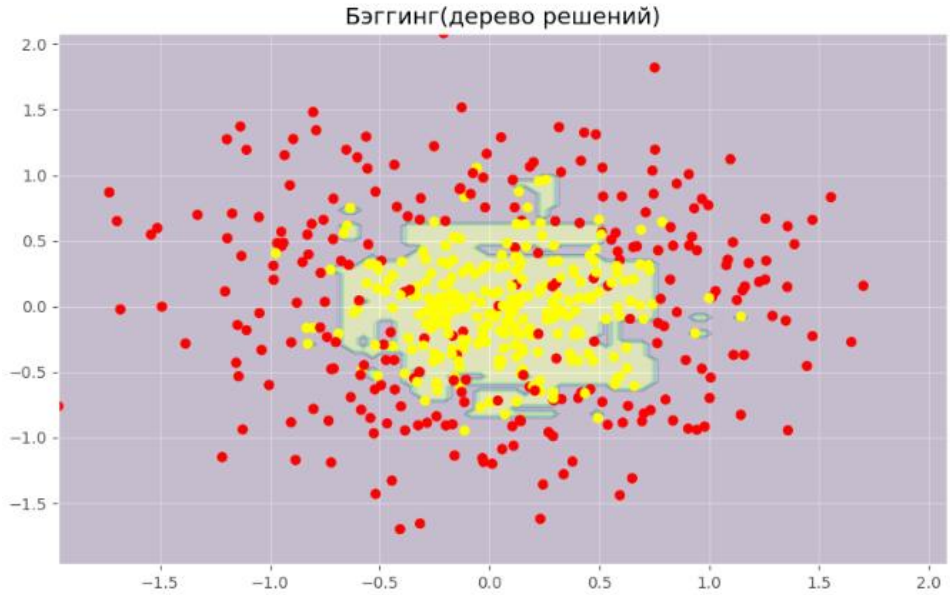


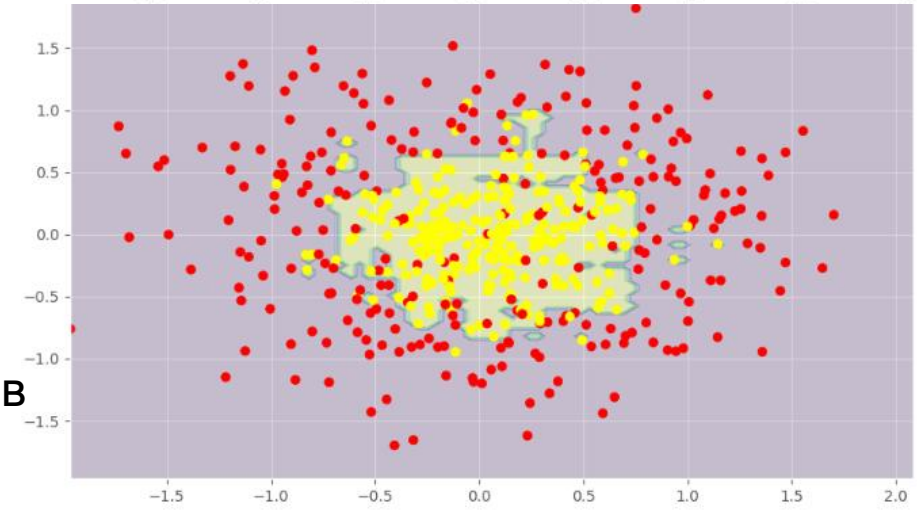
Сравнения решающего дерева, бэггинга и случайного леса в задаче классификации

```
import matplotlib.pyplot as plt
import numpy as np; np.random.seed(42)
from sklearn.ensemble import RandomForestClassifier, BaggingClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.datasets import make_circles  # Make a large circle containing a smaller circle in 2d (toy dataset)
from sklearn.model_selection import train_test_split
plt.style.use('ggplot')
plt.rcParams['figure.figsize'] = 10, 6

# X - координаты центров кругов, y={0;1} - метки классов -> бинарная классификция
X, y = make_circles(n_samples=500, factor=0.1, noise=0.35, random_state=42)
X_train_circles, X_test_circles, y_train_circles, y_test_circles = train_test_split(X, y, test_size=0.2)

dtree = DecisionTreeClassifier(random_state=42)
dtree.fit(X_train_circles, y_train_circles)

# Получаем новые центры кругов
x_range = np.linspace(X.min(), X.max(), 100)
xx1, xx2 = np.meshgrid(x_range, x_range)  # xx1.shape=(100,100),  xx2.shape=(100,100)

# Классифицируем круги
y_hat = dtree.predict(np.c_[xx1.ravel(), xx2.ravel()])  # y_hat.shape = 10000
y_hat = y_hat.reshape(xx1.shape)  # y_hat.shape = (100, 100)

# Отрисовка кругов
plt.contourf(xx1, xx2, y_hat, alpha=0.2)  # Отрисовать область, разграничивающую два класса
plt.scatter(X[:,0], X[:,1], c=y, cmap='autumn')  # Отрисовать круги
plt.title("Дерево решений")
plt.show()

b_dtree = BaggingClassifier(DecisionTreeClassifier(),n_estimators=300, random_state=42)
b_dtree.fit(X_train_circles, y_train_circles)

x_range = np.linspace(X.min(), X.max(), 100)
xx1, xx2 = np.meshgrid(x_range, x_range)
y_hat = b_dtree.predict(np.c_[xx1.ravel(), xx2.ravel()])
y_hat = y_hat.reshape(xx1.shape)
plt.contourf(xx1, xx2, y_hat, alpha=0.2)
plt.scatter(X[:,0], X[:,1], c=y, cmap='autumn')
plt.title("Бэггинг(дерево решений)")
plt.show()

rf = RandomForestClassifier(n_estimators=300, random_state=42)
rf.fit(X_train_circles, y_train_circles)

x_range = np.linspace(X.min(), X.max(), 100)
xx1, xx2 = np.meshgrid(x_range, x_range)
y_hat = rf.predict(np.c_[xx1.ravel(), xx2.ravel()])
y_hat = y_hat.reshape(xx1.shape)
plt.contourf(xx1, xx2, y_hat, alpha=0.2)
plt.scatter(X[:,0], X[:,1], c=y, cmap='autumn')
plt.title("Случайный лес")
plt.show()
```

Сравнения решающего дерева, бэггинга и случайного леса в задаче классификации







Разделяющая граница дерева решений «рваная» и на ней много острых углов, что говорит о переобучении и слабой обобщающей способности. У бэггинга и случайного леса граница достаточно сглаженная и практически нет признаков переобучения.

Параметры RF

class sklearn.ensemble.RandomForestRegressor(
n_estimators – число деревьев (по дефолту – 10)

criterion – функция, которая измеряет качество разбиения ветки дерева (по дефолту – **"mse"**, так же можно выбрать **"mae"**)

max_features – число признаков, по которым ищется разбиение. Можно указать конкретное число или % или выбрать из: **"auto"** (все признаки), **"sqrt"**, **"log2"**. По дефолту стоит **"auto"**

max_depth – максимальная глубина дерева (по дефолту глубина не ограничена)
min_samples_split – минимальное число объектов, необходимое для разделения внутреннего узла. Можно задать числом или процентом от общего числа объектов (по дефолту – 2)
min_samples_leaf – минимальное число объектов в листе. Можно задать числом или % от общего числа объектов (по дефолту – 1)
min_weight_fraction_leaf – минимальная взвешенная доля от общей суммы весов (всех входных объектов) должна быть в листе (по дефолту имеют одинаковый вес)
max_leaf_nodes – максимальное число листьев (по дефолту нет ограничения)
min_impurity_split – порог для остановки наращивания дерева (по дефолту 1e-7)
bootstrap – применять ли бустрэп для построения дерева (по дефолту True)
oob_score – использовать ли out-of-bag объекты для оценки R^2 (по дефолту False)
n_jobs – количество ядер для построения модели и предсказаний (по дефолту 1, если поставить -1, то будут использоваться все ядра)
random_state – начальное значение для генерации случайных чисел (по дефолту его нет, для воспроизведения результатов нужно указать любое число типа int).
verbose – вывод логов по построению деревьев (по дефолту 0)
warm_start – использовать ли уже натренированную модель (по дефолту False))

Параметры RF

В RandomForestClassifier для задачи классификации все те же параметры, отличаются лишь:

class sklearn.ensemble.RandomForestClassifier(

criterion – по дефолту выбран критерий **"gini"**, можно выбрать **"entropy"**

class_weight – вес каждого класса (по дефолту все веса равны 1, но можно передать словарь с весами, либо явно указать **"balanced"**, тогда веса классов будут равны их исходным частям в генеральной совокупности; также можно указать **"balanced_subsample"**, тогда веса на каждой подвыборке будут меняться в зависимости от распределения классов на этой подвыборке)

Параметры, на которые в первую очередь стоит обратить внимание при построении модели:

- **n_estimators** – число деревьев в "лесу"
- **criterion** – критерий для разбиения выборки в вершине
- **max_features** – число признаков, по которым ищется разбиение
- **min_samples_leaf** – минимальное число объектов в листе
- **max_depth** – максимальная глубина дерева

GridSearchCV

```
from sklearn.model_selection import GridSearchCV
```

estimator – estimator object (Метод, параметры которого настраиваем).

param_grid – dict or list of dictionaries. Словарь партеров со значениями для перебора.

scoring – str, callable, list, tuple or dict, default=None. Оценка модели
n_jobs – int, default=None. Количество задействованных ядер процессора. None==1, -1==все ядра процесора.

pre_dispatch – int, or str, default=n_jobs. Управление количеством задач на параллельном выполнении. None == все задания будет созданы сразу, int == точное количество созданных задач, str == выражение от n_jobs, как в "2 * n_jobs.

cv – int, cross-validation generator or an iterable, default=None.
cross-validation splitting strategy. None == 5-fold cross validation, int == точное количество фолдов (Stratified)KFold, есть и другие …

refit – bool, str, or callable, default=True. Переобучить метод используя лучшие значения параметров.

verbose – int. Количество выводимой информации при обучении.
> 1: отображается время вычисления каждого фолда и каждого значения параметра,
> 2: также отображается оценка;
> 3: индексы фолдов и параметров отображаются вместе со временем начала вычисления.

error_score – 'raise' or numeric, default=np.nan. Значение score (оценки) в случае возникновения ошибки. If set to 'raise', the error is raised. If a numeric value is given, FitFailedWarning is raised.

return_train_score – bool, default=False.If False, the cv_results_ attribute will not include training scores. Computing training scores is used to get insights on how different parameter settings impact the overfitting/underfitting trade-off.

RF

```
import pandas as pd
from sklearn.model_selection import cross_val_score, StratifiedKFold, GridSearchCV, train_test_split
from sklearn.metrics import classification_report
from sklearn.ensemble import RandomForestClassifier

# Загружаем данные
df = pd.read_csv("../../....csv")

# Разделяем на признаки и объекты
X_train, X_test, Y_train, Y_test = train_test_split(df.drop(columns='class'), df['class'],
                                                    test_size=0.3, stratify=df['class'])

X_train = X_train.astype(float)  # Может понадобится и для test тоже

# Инициализируем наш классификатор
rfc = RandomForestClassifier(bootstrap=False, random_state=42, n_jobs=-1, criterion='entropy')

# Задаем словарь для перебора значений параттеров
params = {'n_estimators': [150, 155, 160]}

clf = GridSearchCV(rfc, params, scoring='f1', verbose=50, n_jobs=-1)

clf.fit(X_train, Y_train)
"""
Fitting 5 folds for each of 3 candidates, totalling 15 fits
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 2 concurrent workers.
[Parallel(n_jobs=-1)]: Done   1 tasks      | elapsed:  6.0min
...
[Parallel(n_jobs=-1)]: Done  15 out of  15 | elapsed: 52.3min finished

GridSearchCV(cv=None, error_score=nan,
    estimator=RandomForestClassifier(bootstrap=False, ccp_alpha=0.0,
          class_weight=None, criterion='entropy', max_depth=None, max_features='auto',
          max_leaf_nodes=None, max_samples=None, min_impurity_decrease=0.0, min_impurity_split=None,
          min_samples_leaf=1, min_samples_split=2, min_weight_fraction_leaf=0.0,

          n_estimators=100, n_jobs=None, oob_score=False, random_state=None, verbose=0,
          warm_start=False),
iid='deprecated', n_jobs=1,
param_grid={'n_estimators': 150, 155, 160]},
pre_dispatch='2*n_jobs', refit=True, return_train_score=False,
scoring='f1', verbose=50)
"""

# для предсказания используется классификатор с наилучшим recall
best_clf = clf.best_estimator_

# метрики результата предсказания
Y_pred = best_clf.predict(X_test)
print(classification_report(Y_test, Y_pred))
```

BaggingClassifier

base_estimator – object, default=**None**. **None**==DecisionTreeClassifier.
n_estimators – int, default=10. Количество алгоритмов в ансамбле.
max_samples – int **or** float, default=1.0. Количество экземпляров из X на обучение каждого алгоритма. int == указанное количество.float == max_samples * X.shape[0] samples.
max_features – int **or** float, default=1.0. int == наибольше количество рассматриваемых признаков. float == max_features*X.shape[1] features.
Bootstrap – bool, default=**True**. Применить ли бутсрап при отборе экземпляров выборки.
bootstrap_features – bool, default=**False.** Применить ли бутсрап при отборе признаков.
oob_score – bool, default=**False**. Whether to use out-of-bag samples to estimate the generalization error.
warm_start – bool, default=**False. True** == продолжить обучение пердыдущей итерации, иначе обучение нового ансамбля.
n_jobs – int, default=**None**.
random_state – int, default=**None**
verbose – int, default=0

BaggingClassifier

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report
from sklearn.ensemble import BaggingClassifier
from sklearn.model_selection import GridSearchCV

# Загружаем данные
df = pd.read_csv("../../....csv")

# Разделяем на признаки и объекты
X_train, X_test, Y_train, Y_test = train_test_split(df.drop(columns='class'), df['class'],
                                                    test_size=0.3, stratify=df['class'])

X_train = X_train.astype(float)  # Может понадобится и для test тоже

# Инициализируем наш классификатор
bbc = BaggingClassifier(random_state=42)

# Задаем словарь для перебора значений параттеров
params = {'n_estimators': [110], 'bootstrap': [False]}

clf = GridSearchCV(bbc, params, scoring='f1', verbose=50, n_jobs=-1)

clf.fit(X_train, Y_train)

# для предсказания используется классификатор с наилучшим recall
best_clf = clf.best_estimator_

# метрики результата предсказания
Y_pred = best_clf.predict(X_test)
print(classification_report(Y_test, Y_pred))
```

Linear Support Vector Classification. LinearSVC

penalty: {‘l1’, ‘l2’}, default=’l2’. Норма штрафа.

loss: {‘hinge’, ‘squared_hinge’}, default=’squared_hinge’.Функция потерь. ‘hinge’ == standard SVM loss (https://en.wikipedia.org/wiki/Hinge_loss), ‘squared_hinge’== square of the hinge loss. The combination of penalty='l1' and loss='hinge' is not supported.

dual: bool, default=True. algorithm to either solve the dual or primal optimization problem. Prefer dual=False when n_samples > n_features.

tol: float, default=1e-4. Допуск для stopping criteria.

C: float,default=1.0. Параметр регуляризации. Степень применения регуляризации пропорциональная C. Must be strictly positive.

multi_class: {‘ovr’, ‘crammer_singer’}, default=’ovr’. Опредлеляет мультклассовую стратегию, если в y более двух классов. "ovr" trains n_classes one-vs-rest classifiers, while "crammer_singer" optimizes a joint objective over all classes. While crammer_singer is interesting from a theoretical perspective as it is consistent, it is seldom used in practice as it rarely leads to better accuracy and is more expensive to compute. If "crammer_singer" is chosen, the options loss, penalty and dual will be ignored.

fit_intercept bool, default=True. Whether to calculate the intercept for this model. If set to false, no intercept will be used in calculations (i.e. data is expected to be already centered).

intercept_scaling: float, default=1. When self.fit_intercept is True, instance vector x becomes [x, self.intercept_scaling], i.e. a “synthetic” feature with constant value equals to intercept_scaling is appended to the instance vector. The intercept becomes intercept_scaling * synthetic feature weight Note! the synthetic feature weight is subject to l1/l2 regularization as all other features. To lessen the effect of regularization on synthetic feature weight (and therefore on the intercept) intercept_scaling has to be increased.

class_weight: dict or ‘balanced’, default=None. Set the parameter C of class i to class_weight[i]*C for SVC. If not given, all classes are supposed to have weight one. The “balanced” mode uses the values of y to automatically adjust weights inversely proportional to class frequencies in the input data as n_samples / (n_classes * np.bincount(y)).

verbose: int, default=0

random_state: int, RandomState instance or None, default=None

max_iter: int, default=1000. The maximum number of iterations to be run.

Linear Support Vector Classification. LinearSVC

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report
from sklearn.svm import LinearSVC
from sklearn.model_selection import GridSearchCV

# Загружаем данные
df = pd.read_csv("../../....csv")


# Разделяем на признаки и объекты
X_train, X_test, Y_train, Y_test = train_test_split(df.drop(columns='class'),
df['class'], test_size=0.3, stratify=df['class'])

X_train = X_train.astype(float)  # Может понадобится и для test тоже

# Инициализируем наш классификатор
svc = LinearSVC(loss='squared_hinge', penalty='l2', dual=False, verbose=50,
random_state=42)
params = {'class_weight': [None, 'balanced']}

clf = GridSearchCV(svc, params, scoring='f1', verbose=50, n_jobs=-1)

clf.fit(X_train, Y_train)

# для предсказания используется классификатор с наилучшим recall
best_clf = clf.best_estimator_

# метрики результата предсказания
Y_pred = best_clf.predict(X_test)
print(classification_report(Y_test, Y_pred))
```

Бустинг –

техника построения ансамблей, в которой модели применяются последовательно, причем следующая модель учится на ошибках предыдущей. Чаще используются те модели, что дают наибольшую ошибку. Модели обучаются на ошибках, совершенных предыдущими, поэтому требуется меньше времени на обучение. Градиентный бустинг — это пример бустинга.

https://neurohive.io/ru/osnovy-data-science/gradientyj-busting/

Light Gradient Boosted Machine (**LightGBM**)

Библиотека с эффективной реализацией алгоритма градиентного бустинга.
https://lightgbm.readthedocs.io/en/latest/

Алгоритм LightGBM. Две ключевые идеи: **GOSS** и **EFB**.
Градиентная односторонняя выборка (GOSS) – модификация градиентного бустинга с фокусированием внимания на примерах, которые приводят к большему градиенту. GOSS исключает значительную долю экземпляров данных с небольшими градиентами и используем только остальные экземпляры для оценки прироста информации.

Exclusive Feature Bundling (объединение взаимоисключающих признаков) EFB, – подход объединения разрежённых (в основном нулевых) взаимоисключающих признаков, таких как категориальные переменные входных данных, закодированные унитарным кодированием. Это тип автоматического подбора признаков.

GOSS и **EFB** могут ускорить время обучения алгоритма до 20 раз при сохранении точности.

pip install lightgbm

https://habr.com/ru/company/skillfactory/blog/530594/

Ансамбль LightGBM для классификации

Функция make_classification() создает синтетическую задачу бинарной классификации с 1000 примерами и 20 входными признаками.

```
# test classification dataset
from sklearn.datasets import make_classification
# define dataset
X, y = make_classification(n_samples=1000, n_features=20,
n_informative=15, n_redundant=5, random_state=7)
```

Оценить LightGBM с помощью повторной стратифицированной k-кратной кросс-валидации с тремя повторами и k, равным 10. На вывод: среднее и стандартное отклонения точности модели по всем повторениям и сгибам.

LightGBM для классификации

```
# evaluate lightgbm algorithm for classification
from numpy import mean
from numpy import std
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from lightgbm import LGBMClassifier
# define dataset
X, y = make_classification(n_samples=1000, n_features=20,
n_informative=15, n_redundant=5, random_state=7)
# define the model
model = LGBMClassifier()
# evaluate the model
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
n_scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv,
n_jobs=-1)
# report performance
print('Accuracy: %.3f (%.3f)' % (mean(n_scores), std(n_scores)))
```

Accuracy: 0.925 (0.031)

LightGBM для классификации

Ансамбль LightGBM подходит для всех доступных данных и позволяет вызвать функцию *predict()*, чтобы сделать прогнозы по новым данным. Пример бинарной классификации:

```
# make predictions using lightgbm for classification
from sklearn.datasets import make_classification
from lightgbm import LGBMClassifier
# define dataset
X, y = make_classification(n_samples=1000, n_features=20,
n_informative=15, n_redundant=5, random_state=7)
model = LGBMClassifier() # define the model
model.fit(X, y)  # fit the model on the whole dataset
# make a single prediction
row = [0.2929949,-4.21223056,-1.288332,-2.17849815,-0.64527665, 2.58097719,
0.28422388, -7.1827928,-1.91211104, 2.73729512, 0.81395695, 3.96973717,
-2.66939799, 3.34692332, 4.19791821, 0.99990998, -0.30201875, -4.43170633,
-2.82646737,0.44916808]
yhat = model.predict([row])
print('Predicted Class: %d' % yhat[0])
```

Out: Predicted Class: 1

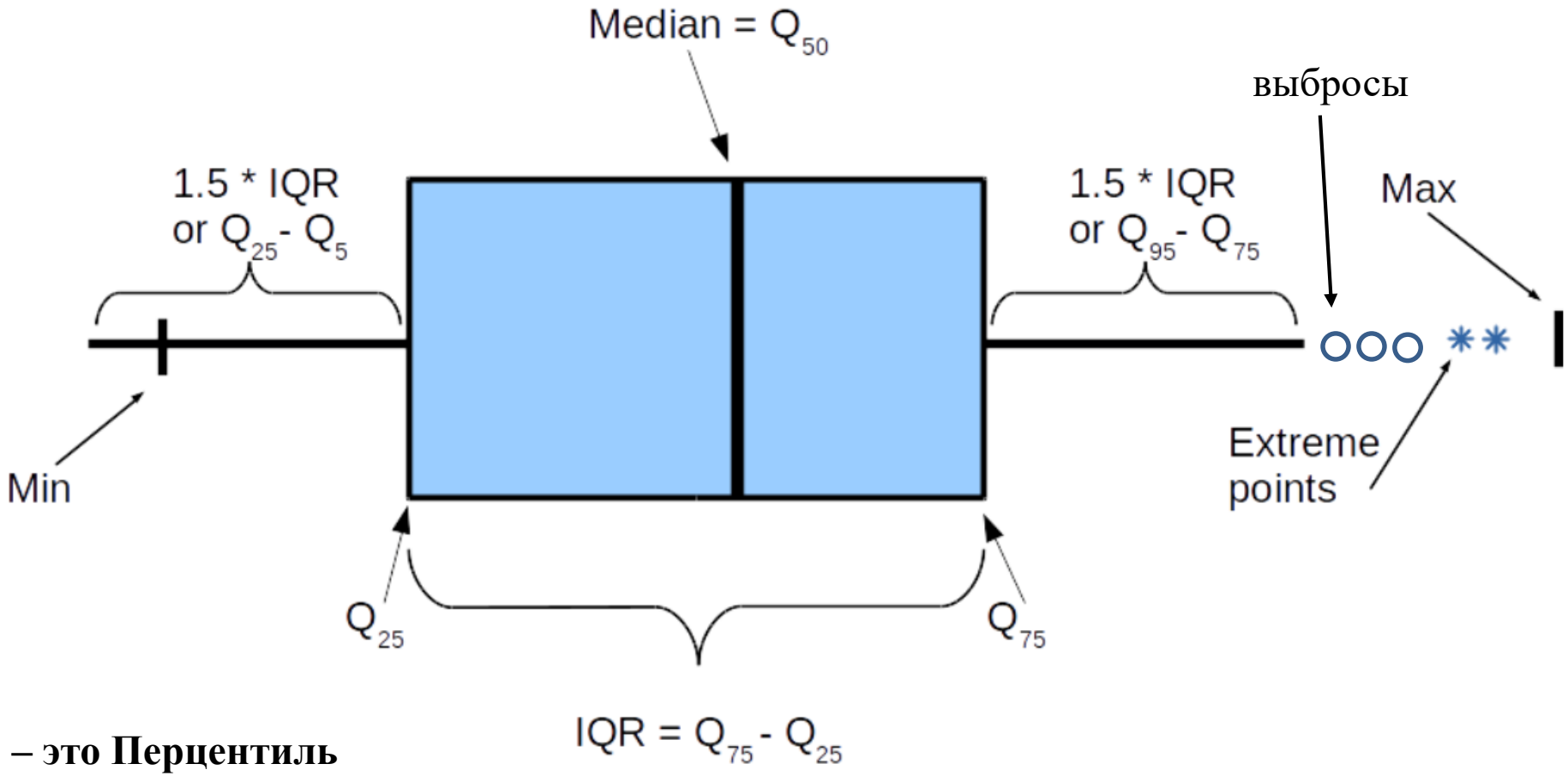
LightGBM. Исследование количества деревьев

Важный гиперпараметр LightGBM – количество деревьев решений в ансамбле. Деревья добавляются в модель последовательно для исправления и улучшения прогнозов, сделанных предыдущими деревьями. Часто работает правило: больше деревьев – лучше. Количество деревьев n_estimators по умолчанию равно 100. В примере исследуется влияние количества деревьев, взяты значения от 10 до 5000:

```
# explore lightgbm number of trees effect on performance
from numpy import mean
from numpy import std
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from lightgbm import LGBMClassifier
from matplotlib import pyplot

# get the dataset
def get_dataset():
    X, y = make_classification(n_samples=1000, n_features=20, n_informative=15, n_redundant=5, random_state=7)
    return X, y

# get a list of models to evaluate
def get_models():
    models = dict()
    trees = [10, 50, 100, 500, 1000, 5000]
    for n in trees:
        models[str(n)] = LGBMClassifier(n_estimators=n)
    return models

# evaluate a give model using cross-validation
def evaluate_model(model):
    cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
    scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
    return scores

# define dataset
X, y = get_dataset()
# get the models to evaluate
models = get_models()
# evaluate the models and store results
results, names = list(), list()
for name, model in models.items():
    scores = evaluate_model(model)
    results.append(scores)
    names.append(name)
    print('>%s %.3f (%.3f)' % (name, mean(scores), std(scores)))
# plot model performance for comparison
pyplot.boxplot(results, labels=names, showmeans=True)
pyplot.show()
```

```
>10 0.859 (0.031)
>50 0.913 (0.027)
>100 0.930 (0.027)
>500 0.940 (0.027)
>1000 0.941 (0.028)
>5000 0.939 (0.027)
```

«Ящик с усами»/диаграммма размахов/boxplot

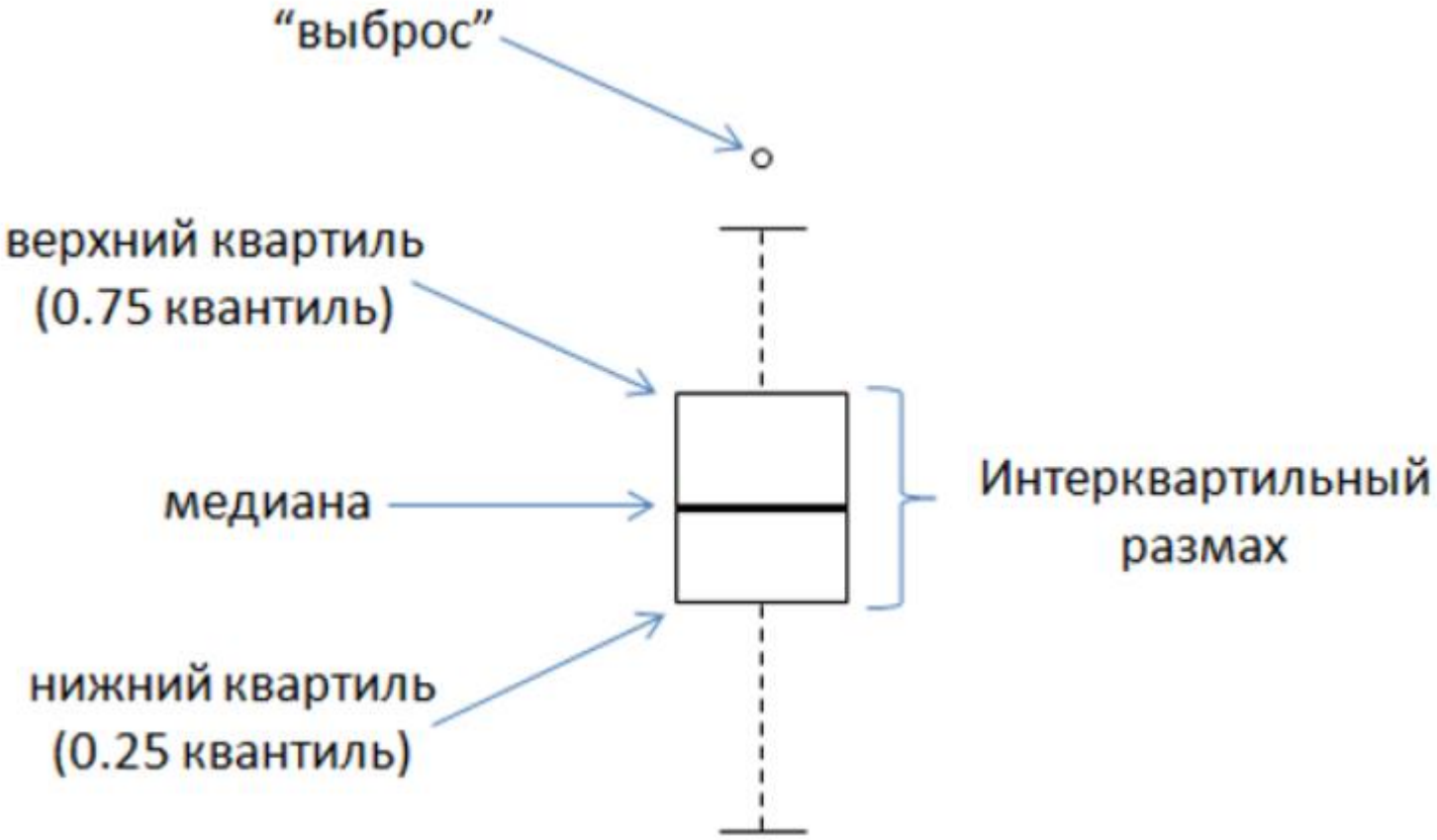


Q – это Перцентиль

Кванти́ль – значение, которое с. в. не превышает с фиксированной вероятностью. Если вероятность задана в %, то квантиль называется **процентилем** или **перцентилем**.

Пример: фраза «90-й процентиль массы тела у новорожденных мальчиков составляет 4 кг» означает, что 90 % мальчиков рождаются с весом, меньшим либо равным 4 кг, а 10 % мальчиков рождаются с весом, большим либо равным 4 кг.

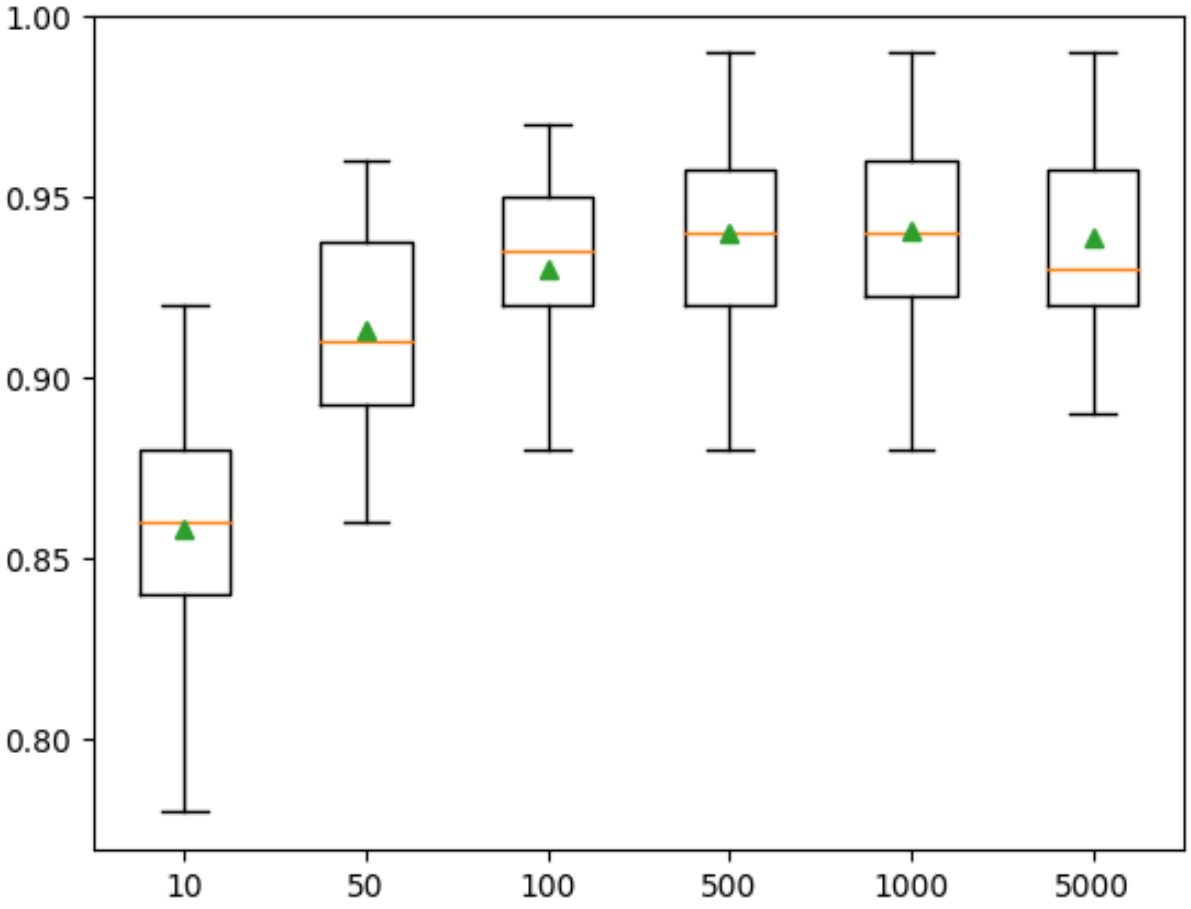
«Ящик с усами»


“выброс”
верхний квартиль
(0.75 квантиль)
медиана
нижний квартиль
(0.25 квантиль)
Интерквартильный
размах

LightGBM. Исследование количества деревьев

>10 0.859 (0.031)
>50 0.913 (0.027)
>100 0.930 (0.027)
>500 0.940 (0.027)
>1000 0.941 (0.028)
>5000 0.939 (0.027)

▲ среднее значение



Исследование скорости обучения, глубины дерева, **типа бустинга**:
https://habr.com/ru/company/skillfactory/blog/530594/

CatBoost

https://catboost.ai/docs/concepts/python-quickstart.html
https://habr.com/ru/company/otus/blog/527554/

pip install catboost

Install visualization tools:

a. Install the ipywidgets Python package (version 7.x or higher is required):

```
pip install ipywidgets
```

b. Turn on the widgets extension:

```
jupyter nbextension enable --py widgetsnbextension
```

CatBoostClassifier

```
import numpy as np
from catboost import CatBoostClassifier, Pool

# initialize data
train_data = np.random.randint(0, 100, size=(100, 10))
train_labels = np.random.randint(0, 2, size=(100))
test_data = catboost_pool = Pool(train_data, train_labels)

model = CatBoostClassifier(iterations=2,
                           depth=2,
                           learning_rate=1,
                           loss_function='Logloss',
                           verbose=True)
# train the model
model.fit(train_data, train_labels)
# make the prediction using the resulting model
preds_class = model.predict(test_data)
preds_proba = model.predict_proba(test_data)
print("class = ", preds_class)
print("proba = ", preds_proba)
```