Практическая справка по тому, как писать код

Содержание

1 DBSCAN: плотностная кластеризация (`sklearn.cluster.DBSCAN`)

Идея. DBSCAN объединяет точки в кластеры там, где *локальная плотность* высокая. Две ключевые константы: ε (`eps`) — радиус окрестности и `min_samples` — минимальное число соседей в этой окрестности. Точки бывают: *core* (ядро, соседей $\geq$`min_samples`), *border* (рядом с ядром, но соседей не хватает) и *noise* (шум, метка -1). Алгоритм не требует заранее задавать число кластеров и умеет находить кластеры произвольной формы.

Когда использовать. Когда ожидаете кластеры произвольной формы и шум; когда масштаб признаков согласован. Плохо работает при резко различной плотности кластеров — тогда смотрите HDBSCAN.

1.1 Базовое применение (с масштабированием)

```python
import numpy as np
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.cluster import DBSCAN

# X: numpy array (N, D)
pipe = make_pipeline(
    StandardScaler(),                       # важен скейлинг
    DBSCAN(eps=0.5, min_samples=10,       # подберите eps и min_samples
           metric="euclidean",              # или "cosine", "manhattan", ...
           algorithm="auto",                # 'auto'/'ball_tree'/'kd_tree'/'brute'
           n_jobs=-1)                       # параллелизм поиска соседей
)

pipe.fit(X)
labels = pipe[-1].labels_                   # метки кластеров, шум == -1
n_clusters = len(set(labels)) - (1 if -1 in labels else 0)
```

Возвращаемые атрибуты (`DBSCAN`):

- `labels_` $(N,)$: целые метки кластеров; -1 — шум.
- `core_sample_indices_`: индексы *core*-точек.
- `components_`: матрица признаков для *core*-точек.

1.2 Ключевые параметры и как их подбирать

- `eps` (> 0): радиус окрестности в *масштабированном* пространстве. Подбирайте через граф *k-distance* (ниже).
- `min_samples` (≥ 1): порог «плотности». Эмпирически: $2 \cdot D$ или $5-10$ для начала.
- `metric`: `"euclidean"` (по умолчанию), `"manhattan"`, `"minkowski"`, `"cosine"`, `"haversine"` и др. Можно передать свою функцию расстояния.
- `p`: степень для `metric="minkowski"`.

- `algorithm`: `"auto"` (выберет сам), `"ball_tree"`, `"kd_tree"`, `"brute"`.
- `leaf_size`: влияет на скорость/память `BallTree`/`KDTree`.
- `n_jobs`: -1 использует все ядра.

k-distance plot: выбор eps. Идея: взять расстояние до k-го соседа (обычно k =`min_samples`), отсортировать и искать «излом» кривой — это и есть разумная ε.

```
import numpy as np
from sklearn.preprocessing import StandardScaler
from sklearn.neighbors import NearestNeighbors
import matplotlib.pyplot as plt

Xs = StandardScaler().fit_transform(X)
k = 10                                      # ~= min_samples
nbrs = NearestNeighbors(n_neighbors=k, n_jobs=-1).fit(Xs)
dists, _ = nbrs.kneighbors(Xs)
kdist = np.sort(dists[:, -1])               # расстояние до k-го соседа
plt.plot(kdist); plt.xlabel("Объекты, отсортированные")
plt.ylabel(f"Расстояние до {k}-го соседа"); plt.show()
# Выберите eps возле "локтя" кривой
```

Оценка кластеризации (с шумом).

```
from sklearn.metrics import silhouette_score

db = DBSCAN(eps=0.35, min_samples=10, n_jobs=-1).fit(Xs)
labels = db.labels_
mask = labels != -1
sil = (silhouette_score(Xs[mask], labels[mask])
       if mask.sum() > 1 and len(np.unique(labels[mask])) > 1 else np.nan)
print("Clusters:", len(set(labels)) - (1 if -1 in labels else 0), "Silhouette:",
    sil)
```

1.3 Косинусная метрика и нормировка

Для текстов/векторов направлений используйте косинусную «дистанцию» ($1-\cos$ine). Перед этим нормируйте векторы до длины 1.

```
from sklearn.preprocessing import normalize
from sklearn.cluster import DBSCAN

Xn = normalize(X, norm="l2")
db = DBSCAN(metric="cosine", eps=0.2, min_samples=5, n_jobs=-1).fit(Xn)
labels = db.labels_
```

1.4 Геоданные: metric=haversine

Для координат (широта/долгота) преобразуйте градусы в *радианы*; eps задаётся тоже в радианах. Например, радиус 500 м $\approx 0.5/6371$ рад.

```python
import numpy as np
from sklearn.cluster import DBSCAN

# coords: (N, 2) в градусах [lat, lon]
coords_rad = np.radians(coords)
eps_km = 0.5  # радиус в км
db = DBSCAN(eps=eps_km/6371.0, min_samples=10,
            metric="haversine", algorithm="ball_tree", n_jobs=-1)
labels = db.fit_predict(coords_rad)
```

1.5 Типичные подводные камни и советы

- **Масштаб признаков.** Без StandardScaler разные масштабы «сломают» расстояния и выбор eps.
- **Разная плотность кластеров.** Один eps может быть «идеален» для одной группы и плох для другой. Рассмотрите HDBSCAN.
- **Выбор min_samples.** Больше значение $\Rightarrow$ кластеры устойчивее к шуму, но больше точек уйдёт в -1.
- **Оценка качества.** silhouette_score считайте только по точкам без шума; для задач с «истинными» метками сравнивайте ARI/NMI.
- **Большие данные.** Для очень больших N следите за памятью (brute может быть дешевле по памяти), подберите leaf_size.

1.6 Постобработка: центры и разбор кластеров

```python
import numpy as np

labels = db.labels_
clusters = [c for c in np.unique(labels) if c != -1]
centroids = {c: X[labels == c].mean(axis=0) for c in clusters}  # "центры масс"
sizes = {c: int((labels == c).sum()) for c in clusters}
noise_share = float((labels == -1).mean())
```

2 Оценка качества кластеризации

Обозначения

Пусть заданы данные $X = \{x_1, \dots, x_N\} \subset \mathbb{R}^d$ и разбиение на K кластеров $\mathcal{C} = \{C_1, \dots, C_K\}$, $C_k \neq \varnothing$, $\bigsqcup_k C_k = X$. Расстояние обозначим $d(\cdot, \cdot)$ (обычно евклидово). Центроид кластера: $\mu_k = \frac{1}{|C_k|} \sum_{x \in C_k} x$. Диаметр кластера: $\Delta(C_k) = \max_{x,y \in C_k} d(x, y)$. Межкластерное расстояние (вариант *single-link*): $\delta(C_i, C_j) = \min_{x \in C_i, y \in C_j} d(x, y)$.

2.1 Внутрикластерное расстояние (compactness)

Мера «сжатости» кластеров. Часто используют сумму квадратов расстояний до центроидов (цель k-means):

$$W(\mathcal{C}) = \sum_{k=1}^{K} \sum_{x \in C_k} \|x - \mu_k\|^2, \qquad \text{или среднюю компактность } \bar{W} = \frac{1}{N} \sum_{k=1}^{K} \sum_{x \in C_k} \|x - \mu_k\|^2.$$

Меньше $W \Rightarrow$ компактнее кластеры. Альтернатива — среднее попарное расстояние внутри каждого кластера: $\frac{2}{|C_k|(|C_k|-1)} \sum_{x \neq y \in C_k} d(x, y)$ и затем среднее по k.

2.2 Межкластерное расстояние (separation)

Характеризует разделённость кластеров. Варианты:

$$B_{\min} = \min_{i \neq j} \delta(C_i, C_j), \qquad B_{\mu} = \min_{i \neq j} d(\mu_i, \mu_j).$$

Больше $B \Rightarrow$ кластеры дальше друг от друга. Часто смотрят *отношение* $\dfrac{B}{\max_k \Delta(C_k)}$ либо $\frac{B}{W}$.

Внутрикластерная «сжатость» и межкластерная «разделённость»: код. Ниже — небольшой набор функций на `NumPy`/`scikit-learn`, который считает: (а) сумму квадратов расстояний до центроидов (compactness, как в k-means), (б) среднее попарное расстояние *внутри* каждого кластера, (в) минимальное расстояние *между* центроидами, и (г) минимальное попарное расстояние между точками разных кластеров. Метка -1 (шум у `DBSCAN`) игнорируется.

```python
import numpy as np
from sklearn.metrics import pairwise_distances

# вспомогатели
def _clusters(labels):
    return [c for c in np.unique(labels) if c != -1]

def _centroids(X, labels):
    cents = {}
    for c in _clusters(labels):
        idx = (labels == c)
        cents[c] = X[idx].mean(axis=0)
    return cents

# ---------- ВНУТРИКЛАСТЕРНЫЕ МЕРЫ (COMPACTNESS) ----------
def within_ssq(X, labels):
    """
    Сумма квадратов расстояний до центроидов:
    total, per_cluster = within_ssq(X, labels)
    """
    cents = _centroids(X, labels)
    total, per_cluster = 0.0, {}
    for c, mu in cents.items():
        idx = (labels == c)
        diff = X[idx] - mu
        val = float((diff * diff).sum())
        per_cluster[c] = val
        total += val
    return total, per_cluster

def within_avg_pairwise(X, labels, metric="euclidean"):
    """
    Среднее попарное расстояние внутри каждого кластера.
    """
    res = {}
    for c in _clusters(labels):
        idx = np.where(labels == c)[0]
        if len(idx) < 2:
            res[c] = 0.0
            continue
        D = pairwise_distances(X[idx], metric=metric)
        res[c] = float(D[np.triu_indices_from(D, k=1)].mean())
    return res

# ---------- МЕЖКЛАСТЕРНЫЕ МЕРЫ (SEPARATION) ----------
def between_min_centroid(X, labels, metric="euclidean"):
    """
    Минимальное расстояние между центроидами (по парам кластеров).
    """
    cents = _centroids(X, labels)
    keys = list(cents.keys())
    if len(keys) < 2:
        return np.nan
    M = np.vstack([cents[k] for k in keys])
    D = pairwise_distances(M, metric=metric)
    return float(D[np.triu_indices_from(D, 1)].min())

def between_min_intercluster(X, labels, metric="euclidean"):
    """
    Минимальное попарное расстояние между точками разных кластеров.
    """
    cs = _clusters(labels)
    if len(cs) < 2:
        return np.nan
    D = pairwise_distances(X, metric=metric)
```

Замечания. (1) Для высоких размерностей или текстовых признаков используйте `metric="cosine"` в обеих межах. (2) `between_min_intercluster` имеет квадратичную сложность по N (через матрицу расстояний) — на очень больших выборках возьмите сэмпл или переходите на эвристики (напр., расстояния между центроидами/медианами). (3) Перед расчётом расстояний масштабируйте признаки (`StandardScaler`), иначе метрики будут доминировать размерностями с крупным масштабом.

2.3 Индекс Данна (Dunn index)

Классическая безучётная метрика «разделённость / рыхлость»:

$$D(\mathcal{C}) = \frac{\min\limits_{i \neq j} \ \delta(C_i, C_j)}{\max\limits_{k} \ \Delta(C_k)} \in (0, \infty).$$

Чем больше D, тем лучше (кластеры далеко и компактны). Чувствителен к выбросам (и в Δ, и в δ); на практике используют робастные варианты (напр., средние радиусы вместо диаметров или δ по центроидам).

Пример вычисления (евклидово расстояние).

```python
import numpy as np
from sklearn.metrics import pairwise_distances

def dunn_index(X, labels, metric="euclidean"):
    X = np.asarray(X)
    D = pairwise_distances(X, metric=metric)
    clusters = [np.where(labels==c)[0] for c in np.unique(labels) if c != -1]  # игнор шум
    if len(clusters) < 2:
        return np.nan
    intra = []
    for idx in clusters:
        if len(idx) < 2: intra.append(0.0)
        else: intra.append(D[np.ix_(idx, idx)][np.triu_indices(len(idx), 1)].max())
    inter = []
    for i in range(len(clusters)):
        for j in range(i+1, len(clusters)):
            inter.append(D[np.ix_(clusters[i], clusters[j])].min())
    return (np.min(inter) / np.max(intra)) if np.max(intra) > 0 else np.inf
```

2.4 Силуэт (Silhouette)

Для каждой точки x_i:

$$a(i) = \frac{1}{|C(x_i)| - 1} \sum_{x \in C(x_i),\, x \neq x_i} d(x_i, x) \quad \text{— среднее расстояние до «своего» кластера,}$$

$b(i) = \min\limits_{k \neq C(x_i)} \dfrac{1}{|C_k|} \sum\limits_{x \in C_k} d(x_i, x)$ — наименьшее среднее расстояние до «чужих» кластеров,

$$s(i) = \frac{b(i) - a(i)}{\max\{a(i),\, b(i)\}} \in [-1,\, 1].$$

Интерпретация: $s(i) \approx 1$ — точка хорошо «внутри» своего кластера; $s(i) \approx 0$ — на границе; $s(i) < 0$ — вероятно, присвоена неверно. Глобальный силуэт — среднее $S = \frac{1}{N} \sum_i s(i)$ (чем больше, тем лучше).

Код (`sklearn`).

```python
import numpy as np
from sklearn.metrics import silhouette_score, silhouette_samples

# labels: метки кластеров (шум у DBSCAN помечают -1, его часто исключают)
mask = labels != -1
S = (silhouette_score(X[mask], labels[mask], metric="euclidean")
     if mask.sum() > 1 and len(np.unique(labels[mask])) > 1 else np.nan)
# np.isfinite(S) проверяет, число конечное
s_i = (silhouette_samples(X[mask], labels[mask], metric="euclidean")
       if np.isfinite(S) else None)
```

2.5 Практические замечания

- **Масштабирование.** Метрики расстояния чувствительны к масштабу. Перед расчётом метрик и обучением кластеризаторов используйте `StandardScaler` (или `Normalizer` при косинусной метрике).
- **Шум/выбросы.** Для DBSCAN/HDBSCAN метки -1 (*noise*) обычно исключают из силуэта и индекса Данна, иначе метрики искажаются.
- **Несбалансированные кластеры.** Силуэт может «наказывать» сильно разных по размеру/плотности кластеров; смотрите совместно несколько метрик и визуализацию.
- **Выбор K.** Для k-means число кластеров нередко выбирают по максимуму среднего силуэта или по «локтю» кривой $W(K)$.
- **Метрика d.** В высоких размерностях и для текстов лучше использовать косинусную меру.

3 Регрессии в `scikit-learn`: базовые модели и параметры

В этом разделе — минимальные рабочие шаблоны кода для часто используемых регрессионных моделей и краткие пояснения параметров. Теория (оценка МНК, L_1/L_2-штрафы, байесовские интерпретации) — в отдельном файле.

3.1 Обычная линейная регрессия (OLS, `LinearRegression`)

Мини-пример.

```python
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score

# X: (n_samples, n_features), y: (n_samples,)
Xtr, Xte, ytr, yte = train_test_split(X, y, test_size=0.2, random_state=42)

ols = LinearRegression(      # МНК без регуляризации
    fit_intercept=True,      # обучать свободный член
    copy_X=True,             # делать копию X (безопаснее, но медленнее по памяти)
    n_jobs=None,             # многопроцессорность (некоторые сборки sklearn её игнорируют)
    positive=False           # ограничить коэффициенты >= 0
)
ols.fit(Xtr, ytr)

yhat = ols.predict(Xte)
print("MSE:", mean_squared_error(yte, yhat))
print("R2 :", r2_score(yte, yhat))
```

Ключевые параметры.

- `fit_intercept` — добавлять ли константный признак (свободный член). Если данные уже центрированы, можно снять.
- `copy_X` — копировать ли входную матрицу; `False` экономит память, но может менять `X` внутри.
- `positive` — ограничивать коэффициенты неотрицательностью (квадратичная задача с ограничениями).
- R^2`-score` показывает, насколько хорошо модель регрессии объясняет вариацию целевой переменной.

Когда применять. Быстрый и интерпретируемый базовый уровень без регуляризации; чувствителен к мультиколлинеарности и выбросам.

3.2 Ridge-регрессия (`Ridge`, L_2-штраф)

Мини-пример.

```python
from sklearn.linear_model import Ridge

ridge = Ridge(
    alpha=1.0,               # сила L2 (больше -> сильнее сглаживание весов)
    fit_intercept=True,
    solver="auto",           # "auto", "svd", "cholesky", "lsqr", "sparse_cg",
    "sag", "saga", "lbfgs"
    random_state=42,         # влияет для стохастических солверов
    max_iter=None, tol=1e-4
)
ridge.fit(Xtr, ytr)
yhat = ridge.predict(Xte)
```

Ключевые параметры.

- `alpha` — коэффициент L_2-регуляризации: подавляет раздувание коэффициентов при мультиколлинеарности.
- `solver` — выбор алгоритма решения: `"sag"/"saga"` годятся для больших и разреженных задач; `"svd"` — устойчиво при коллинеарности; `"lbfgs"` — квази-Ньютон.
- `max_iter` — лимит итераций
- `tol` — точность сходимости для итеративных солверов (Алгоритм итеративно обновляет веса до тех пор, пока изменения между шагами не станут меньше заданного порога tol).

Отличия от OLS. Добавляет L_2-штраф, уменьшает дисперсию оценок и переобучение, но не зануляет коэффициенты (все признаки остаются).

3.3 Lasso-регрессия (`Lasso`, L_1-штраф)

Мини-пример.

```python
from sklearn.linear_model import Lasso

lasso = Lasso(
    alpha=1e-3,              # сила L1 (больше -> больше нулей в коэффициентах)
    fit_intercept=True,
    max_iter=5000, tol=1e-4,
    selection="cyclic",      # "cyclic" или "random" координатный спуск
    warm_start=False,
    random_state=42
)
lasso.fit(Xtr, ytr)
yhat = lasso.predict(Xte)
```

Ключевые параметры.

- `alpha` — коэффициент L_1-штрафа: стимулирует разреженные решения (feature selection).
- `selection` — порядок обновления координат: `"cyclic"` быстрее на плотных данных, `"random"` — иногда стабильнее.
- `max_iter`, `tol` — контроль сходимости координатного спуска.
- При `warm_start=True`: новые итерации стартуют с предыдущего решения → можно дообучать модель или быстрее решать похожие задачи (`False` означает, что каждый вызов .fit() начинает обучение "с нуля").

Отличия от Ridge. L_1 может занулять коэффициенты и фактически отбирать признаки; более чувствителен к масштабам — *масштабируйте* признаки.

3.4 Коротко о различиях: OLS vs Ridge vs Lasso

- **OLS** (`LinearRegression`) — без штрафа; минимальная смещение, максимум дисперсии при мультиколлинеарности.
- **Ridge** (`Ridge`, L_2) — сглаживает веса, снижает дисперсию, не обнуляет коэффициенты.
- **Lasso** (`Lasso`, L_1) — делает веса разреженными, выполняет отбор признаков, но даёт больший сдвиг (bias).

3.5 Логистическая регрессия (`LogisticRegression`)

Мини-пример (бинарная классификация).

```
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import make_pipeline
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import log_loss, accuracy_score, roc_auc_score

pipe = make_pipeline(
    StandardScaler(),
    LogisticRegression(
        solver="lbfgs",         # устойчивый квази-Ньютон
        penalty="l2",           # тип регуляризации
        C=1.0,                  # обратная сила L2 (меньше C -> сильнее штраф)
        max_iter=200,
        class_weight=None,      # или "balanced" при дисбалансе
        random_state=42
    )
)
pipe.fit(Xtr, ytr)
proba = pipe.predict_proba(Xte)[:, 1] # возращает вероятность принадлежности к п
    ервому классу
print("LogLoss:", log_loss(yte, proba))
print("AUC    :", roc_auc_score(yte, proba))
print("Acc    :", accuracy_score(yte, proba >= 0.5))
```

Ключевые параметры.

- `solver` — `"lbfgs"` (универсальный), `"liblinear"` (малые данные, бинарная), `"saga"` (большие/разреженные, поддерживает `l1`/`elasticnet`).
- `penalty` — `"l2"`, `"l1"` (с `saga`/`liblinear`), `"elasticnet"` (требует `saga` + `l1_ratio`).
- `C` — обратная сила штрафа: меньше `C` ⇒ сильнее регуляризация.
- `max_iter`, `tol` — контроль сходимости оптимизатора.
- `class_weight` — балансировка классов (`"balanced"` или словарь весов).
- `multi_class` — `"auto"` (OvR/softmax по ситуации), `"ovr"`, `"multinomial"` (с `lbfgs`/`saga`).

Когда применять. Быстрый, устойчивый и интерпретируемый линейный классификатор; используйте масштабирование и регуляризацию.

Логистическая регрессия для многоклассовой классификации (softmax)

```python
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import make_pipeline
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, log_loss, classification_report

# 1) Данные: 3 класса (Iris)
X, y = load_iris(return_X_y=True)
Xtr, Xte, ytr, yte = train_test_split(
    X, y, test_size=0.25, random_state=42, stratify=y
)

# 2) Пайплайн: масштабирование + многоклассовая логистическая (softmax)
pipe = make_pipeline(
    StandardScaler(),
    LogisticRegression(
        multi_class="multinomial",    # softmax (а не one-vs-rest)
        solver="lbfgs",               # поддерживает 'multinomial'
        C=1.0,                        # сила L2 (меньше C -> сильнее регуляризаци
    я)
        max_iter=200,                 # лимит итераций оптимизатора
        random_state=42
    )
)

# 3) Обучение и оценка
pipe.fit(Xtr, ytr)
proba = pipe.predict_proba(Xte)       # shape: (n_test, K)
ypred = pipe.predict(Xte)

print("Accuracy:", accuracy_score(yte, ypred))
print("LogLoss:", log_loss(yte, proba))
print(classification_report(yte, ypred))
```

Замечания.

- `multi_class="multinomial"` включает настоящую softmax-регрессию; альтернатива `"ovr"` обучает K бинарных задач one-vs-rest.
- `solver="lbfgs"` подходит для плотных данных и поддерживает `multinomial`; для больших/разреженных матриц рассмотрите `solver="saga"`.
- `C` — обратная сила L_2-регуляризации: уменьшение `C` усиливает регуляризацию.
- `predict_proba` возвращает матрицу $(N_{\text{test}} \times K)$ в порядке `clf.classes_`; строки суммируются к 1.
- `classification_report` — это удобная функция из библиотеки scikit-learn (sklearn.metrics), которая выводит основные метрики качества классификации для каждой категории (класса).

3.6 Какие ещё регрессии есть в `sklearn` (куда смотреть)

- **ElasticNet** (`ElasticNet`) — комбинация L_1 и L_2; ключевые параметры: `alpha`, `l1_ratio`.
- **HuberRegressor** — робастная к выбросам (квадратичная в малых остатках, линейная в больших); `epsilon` задаёт порог.
- **RANSACRegressor** — робастная аппроксимация через повторное подвыборочное оценивание; хорошо при большом числе выбросов.
- **QuantileRegressor** — оптимизация по квантильной потере (медианная/квантильная регрессия) для асимметричных распределений ошибок.
- **PoissonRegressor**, **GammaRegressor**, **TweedieRegressor** — обобщённые линейные модели для положительных счётных/неотрицательных целевых переменных.
- **SVR** — опорные векторы для регрессии (ядра: `linear`, `rbf`, `poly`); параметры: `C`, `epsilon`, `gamma`.
- **Деревья/ансамбли**: `DecisionTreeRegressor`, `RandomForestRegressor`, `ExtraTreesRegressor`, `HistGradientBoostingRegressor` — нелинейные, хорошо работают по умолчанию, меньше требуют препроцессинга.

Практические заметки.

- Для **Lasso/ElasticNet** всегда масштабируйте признаки (`StandardScaler`) — штрафы чувствительны к масштабу.
- **Ridge** устойчив к мультиколлинеарности и часто даёт лучшее обобщение, чем OLS.
- Подбирайте `alpha`/`C` по CV (`RidgeCV`, `LassoCV`, `LogisticRegressionCV` или `GridSearchCV`/`Random`

Примеры кода с ядрами (SVM/SVR, Kernel Ridge, предвычисленное ядро, аппроксимации, Kernel PCA)

SVC с RBF-ядром и подбором гиперпараметров.

```python
from sklearn.svm import SVC
from sklearn.model_selection import GridSearchCV, StratifiedKFold
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import make_pipeline

pipe = make_pipeline(
    StandardScaler(),
    SVC(kernel="rbf", probability=False)
)

param_grid = {
    "svc__C": [0.1, 1, 10, 100],
    "svc__gamma": [1e-3, 1e-2, 1e-1, 1.0]
}

cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
gs = GridSearchCV(
    pipe, param_grid=param_grid, cv=cv,
    scoring="roc_auc", n_jobs=-1, refit=True
)
gs.fit(X_train, y_train)

print("Best params:", gs.best_params_)
print("Best CV AUC:", gs.best_score_)
y_pred = gs.predict(X_test)
```

Комментарии.

- `kernel="rbf"` — универсальный вариант для нелинейных границ; требуется масштабирование признаков.
- `C` контролирует штраф за ошибки, `gamma` — «радиус влияния» точки (меньше — гладче).
- `StratifiedKFold` сохраняет доли классов по фолдам; `scoring="roc_auc"` — порогонезависимая метрика.
- `GridSearchCV` — это инструмент, который ищет оптимальные гиперпараметры модели перебором по сетке значений.

Линейный SVM: `SVC(kernel="linear")` **и** `LinearSVC`.

```python
from sklearn.svm import SVC, LinearSVC
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import make_pipeline

# Вариант 1: SVC с линейным ядром (поддерживает probability=True)
svc_lin = make_pipeline(
    StandardScaler(),
    SVC(kernel="linear", C=1.0)
).fit(X_train, y_train)

# Вариант 2: LinearSVC (быстрее на больших n_features, без predict_proba)
lsvc = make_pipeline(
    StandardScaler(),
    LinearSVC(C=1.0, dual="auto", max_iter=5000)
).fit(X_train, y_train)

scores_svc = svc_lin.decision_function(X_test)  # real-valued scores
preds_lsvc = lsvc.predict(X_test)
```

Комментарии.

- `SVC(kernel="linear")` хранит опорные вектора; `LinearSVC` решает линейную задачу без явных ОВ (масштабируется лучше).
- `dual="auto"` выбирает форму задачи в зависимости от соотношения $n\_samples$ и $n\_features$.

SVR с RBF-ядром для регрессии.

```python
from sklearn.svm import SVR
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import make_pipeline
from sklearn.metrics import mean_absolute_error

svr = make_pipeline(
    StandardScaler(),
    SVR(kernel="rbf", C=10.0, epsilon=0.1, gamma=1e-2)
).fit(X_train, y_train)

y_pred = svr.predict(X_test)
print("MAE:", mean_absolute_error(y_test, y_pred))
```

Комментарии.

- `epsilon` — ширина «трубки» без штрафа; повышает устойчивость к шуму.
- `C` и `gamma` — аналогично SVC с RBF.

SVC с полиномиальным ядром.

```python
from sklearn.svm import SVC
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import make_pipeline

svc_poly = make_pipeline(
    StandardScaler(),
    SVC(kernel="poly", degree=3, gamma="scale", coef0=1.0, C=1.0)
).fit(X_train, y_train)

y_pred = svc_poly.predict(X_test)
```

Комментарии.

- `degree` — степень полинома (обычно 2–3); `coef0` добавляет сдвиг (важен при малых степенях).
- `gamma` масштабирует вклад признаков в полиномиальное ядро.

Kernel Ridge Regression (KRR) с RBF-ядром.

```python
from sklearn.kernel_ridge import KernelRidge
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import make_pipeline

krr = make_pipeline(
    StandardScaler(),
    KernelRidge(kernel="rbf", alpha=1.0, gamma=1e-2)
).fit(X_train, y_train)

y_pred = krr.predict(X_test)
```

Комментарии.

- `alpha` — коэффициент L2-регуляризации в решении; `gamma` — как в RBF.
- Часто даёт гладкие аппроксимации; масштабируется по n как ядровые методы.

SVC с предвычисленным ядром (пример на косинусном сходстве).

```python
import numpy as np
from sklearn.svm import SVC
from sklearn.metrics.pairwise import cosine_similarity

# K_train: (n_train x n_train), K_test: (n_test x n_train)
K_train = cosine_similarity(X_train, X_train)
K_test  = cosine_similarity(X_test,  X_train)

svc_pre = SVC(kernel="precomputed", C=1.0).fit(K_train, y_train)
y_pred = svc_pre.predict(K_test)
```

Комментарии.

- Для `kernel="precomputed"` на трейне подаётся квадратная симметричная матрица сходства; на тесте — прямоугольная ($n\_test \times n\_train$).
- Проследите, чтобы ядро было положительно полуопределённым (иначе возможны нестабильности).

4 PCA: практическое использование и подбор параметров

Идея и когда применять. Principal Component Analysis (PCA) понижает размерность данных, поворачивая признаки в новые ортогональные оси (главные компоненты), ранжированные по убыванию объяснённой дисперсии. Применяйте для сжатия признаков, удаления корреляций, ускорения моделей/визуализаций и борьбы с шумом. Всегда масштабируйте признаки перед PCA.

Основные параметры `sklearn.decomposition.PCA`.

- `n_components`:
  - целое d — число главных компонент (ГК);
  - дробь $0 < \alpha \leq 1$ — держать долю дисперсии $\geq \alpha$ (напр., `0.95`);
  - `"mle"` — Minka MLE (автовыбор размерности; `svd_solver="full"`);
  - `None` — взять все компоненты.
- `whiten` (bool): «выбеливание» — нормирует проекции к единичной дисперсии. Может помочь некоторым моделям, но усиливает шум.
- `svd_solver`: `"auto"`, `"full"` (точный SVD), `"arpack"` (когда нужно мало компонент), `"randomized"` (быстрый приближённый для больших матриц).
- `random_state`: сид для `"randomized"`.
- `iterated_power`, `tol`: тонкая настройка для `"randomized"`/`"arpack"`.

Шаги подбора (интуитивно).

1. Постройте *scree plot*: по оси x — индекс ГК, по оси y — `explained_variance_` (или доля `explained_variance_ratio_`). Ищите «локоть».
2. Постройте накопленную долю дисперсии $\sum_{j \leq d} \texttt{explained\_variance\_ratio\_}[j]$ и выберите d при желаемом пороге (например, $\geq 80\%$).
3. Для визуализации обычно берут $d = 2$ или $d = 3$.
4. Если есть предметные знания — задайте d явно.
5. В составе модели рассматривайте d как гиперпараметр и подбирайте его по метрике качества на CV.

Базовый шаблон: масштабирование + PCA.

```python
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import make_pipeline

pca = make_pipeline(
    StandardScaler(),
    PCA(n_components=2, svd_solver="auto", whiten=False, random_state=42)
)
X_pca = pca.fit_transform(X)  # форма: (n_samples, 2)
```

Комментарии.

- `StandardScaler` обязателен: PCA чувствителен к масштабу признаков.
- `whiten=False` по умолчанию: включает `whiten=True` только при необходимости.

Графики «каменистой осыпи» и накопленной доли.

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler

Xz = StandardScaler().fit_transform(X)
pca_full = PCA(n_components=None, svd_solver="full").fit(Xz)

ev = pca_full.explained_variance_
evr = pca_full.explained_variance_ratio_
cev = np.cumsum(evr)

plt.figure(figsize=(6,3.2))
plt.plot(np.arange(1, len(ev)+1), ev, marker="o")
plt.xlabel("Номер компоненты"); plt.ylabel("Собственное значение")
plt.title("Scree plot"); plt.tight_layout(); plt.show()

plt.figure(figsize=(6,3.2))
plt.plot(np.arange(1, len(cev)+1), cev, marker="o")
plt.axhline(0.95, ls="--")  # порог 95%
plt.xlabel("d"); plt.ylabel("Накопленная доля дисперсии")
plt.title("Explained variance (cumulative)"); plt.tight_layout(); plt.show()
```

Автовыбор размерности (`n_components="mle"` или доля дисперсии).

```python
# 1) MLE (автовыбор d): работает с svd_solver="full"
Xz = StandardScaler().fit_transform(X)
pca_mle = PCA(n_components="mle", svd_solver="full").fit(Xz)
X_mle = pca_mle.transform(Xz)  # d выбрано автоматически

# 2) По доле дисперсии (напр., 95%)
pca_95 = PCA(n_components=0.95, svd_solver="full").fit(Xz)
X_95 = pca_95.transform(Xz)   # число компонент выбрано под порог 0.95
```

Подбор `n_components` по метрике модели (GridSearchCV).

```python
from sklearn.model_selection import GridSearchCV, StratifiedKFold
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.decomposition import PCA

pipe = Pipeline([
    ("sc", StandardScaler()),
    ("pca", PCA(svd_solver="auto", whiten=False, random_state=42)),
    ("clf", LogisticRegression(solver="lbfgs", max_iter=1000, random_state=42))
])

param_grid = {
    "pca__n_components": [2, 5, 10, 20, 0.95],  # int или доля дисперсии
    "pca__whiten": [False, True]
}

cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
gs = GridSearchCV(
    pipe, param_grid=param_grid,
    scoring="roc_auc", cv=cv, n_jobs=-1
)
gs.fit(X, y)

print("Best params:", gs.best_params_)
print("Best score:", gs.best_score_)
```

Важно.

- PCA включён в *Pipeline* — так мы избегаем утечек: стандартизация и PCA переобучаются только на фолдах тренировки внутри CV.
- В `param_grid` параметры адресуются как `"pca__..."`.

Ручной перебор `n_components` (быстрый baseline).

```python
from sklearn.model_selection import cross_val_score
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.linear_model import LogisticRegression
import numpy as np

ds = [2, 5, 10, 20]
scores = []
for d in ds:
    pipe = make_pipeline(
        StandardScaler(),
        PCA(n_components=d, svd_solver="auto"),
        LogisticRegression(solver="lbfgs", max_iter=1000, random_state=7)
    )
    s = cross_val_score(pipe, X, y, scoring="roc_auc", cv=5, n_jobs=-1)
    scores.append((d, np.mean(s), np.std(s)))

for d, m, s in scores:
    print(f"d={d:2d} | AUC={m:.3f}  {s:.3f}")
```

Частые ошибки и советы.

- **Не масштабировали** признаки — первая компонента «украдёт» самый крупномасштабный признак.
- Делаете PCA *до* разбиения на фолды — это утечка; всегда используйте `Pipeline`.
- Без нужды включили `whiten=True` — может ухудшить устойчивость/шум; включайте осознанно.
- Для разреженных матриц используйте `TruncatedSVD`, а не `PCA`.
- Проверяйте `explained_variance_ratio_` и её сумму: это основной ориентир выбора d.

5 Ансамбли: практические рецепты с кодом

5.1 Бэггинг: Random Forest

Идея. Бэггинг строит множество независимых деревьев на бутстрэп-выборках и усредняет их предсказания (голосование/среднее). Это снижает дисперсию модели.

Ключевые параметры `RandomForestClassifier/Regressor`.

- `n_estimators`: число деревьев (обычно 200–1000).
- `max_depth`: ограничение глубины (контроль переобучения).
- `max_features`: число признаков на узел (`"sqrt"` для классификации, `"log2"` или доля для числовых).
- `min_samples_split`, `min_samples_leaf`: минимумы для разбиений/листьев.
- `bootstrap=True`: обучение на бутстрэп-выборках; `oob_score=True` — OOB-оценка качества.
- `class_weight`: балансировка классов (`"balanced"` при дисбалансе).
- `random_state`, `n_jobs`: воспроизводимость и параллельность.

Мини-код (классификация и регрессия).

```python
from sklearn.ensemble import RandomForestClassifier, RandomForestRegressor
from sklearn.model_selection import cross_val_score

rf_clf = RandomForestClassifier(
    n_estimators=500, max_depth=None, max_features="sqrt",
    min_samples_leaf=1, bootstrap=True, oob_score=True,
    n_jobs=-1, random_state=7
).fit(Xtr, ytr)

rf_reg = RandomForestRegressor(
    n_estimators=600, max_depth=None, max_features=0.7,
    min_samples_leaf=2, bootstrap=True, n_jobs=-1, random_state=7
).fit(Xtr_reg, ytr_reg)

# Функция получает у объекта(rf_clf) значение атрибута с названием "oob_score_"
print("OOB:", getattr(rf_clf, "oob_score_", None))
print("CV AUC:", cross_val_score(rf_clf, Xtr, ytr, cv=5,
    scoring="roc_auc").mean())
```

Тюнинг (кратко). Начните с `n_estimators` достаточно большим, затем подбирайте `max_features` и `min_samples_leaf`. Для дисбаланса включайте `class_weight="balanced"`.

5.2 Градиентный бустинг по деревьям: CatBoost, XGBoost, LightGBM

Общее. Бустинг строит деревья последовательно, каждое исправляет ошибки предыдущих. Ключевые «рычаги»: `learning_rate` (шаг), сложность дерева (`depth`/`max_depth`/`num_leaves`),

регуляризация и стохастика (`subsample`, `colsample`).

5.2.1 CatBoost (классификация/регрессия)

Сильные стороны: нативная категорика, ordered boosting, разумные дефолты.

Главные параметры.

- `iterations` (число деревьев), `learning_rate` (обычно 0.02–0.2), `depth` (4–10).
- `l2_leaf_reg` (L2 на листья), `random_strength`, `bagging_temperature`, `rsm` (кол-во фич).
- `loss_function` (логлосс/MAE/MSE и т. п.), `eval_metric`, `early_stopping_rounds`.
- `cat_features`: индексы категориальных колонок.

Что такое `bagging_temperature`? Параметр управляет *силой стохастики* при обучении через байесовский бутстрэп (перевзвешивание объектов вместо обычного подвыборивания).

- 0.0 — почти детерминированное обучение (минимум шума); выше риск переобучения на сложных данных.
- $0.1 \dots 1.0$ — умеренная стохастика (часто лучшие компромиссы по качеству/устойчивости).
- >1.0 — сильная стохастика (агрессивное «разнообразие» деревьев), полезно на больших/шумных датасетах, но может ухудшать сходимость.

Интуитивно: чем выше `bagging_temperature`, тем более «шероховатыми» становятся веса объектов в каждом бустинговом шаге.

Что такое `rsm` (Random Subspace Method)? Это доля признаков, которая случайно отбирается для *каждого сплита* дерева.

- Значения: $0.1 \leq$ `rsm` ≤ 1.0 (по умолчанию 1.0 — все признаки).
- **Зачем:** снижает коррелированность деревьев и риск переобучения, ускоряет обучение на высокоразмерных данных.
- **Практика:** начните с 0.8–0.95 для числовых данных; при большом числе признаков можно опускаться до 0.5.

Часто используемые методы/атрибуты модели CatBoost.

- `fit(Pool/ (X,y))`, `predict(X)`, `predict_proba(X)` — базовые операции обучения/-инференса.
- `get_best_iteration()`, `get_best_score()` — лучшая итерация и метрики валидации при ранней остановке.
- `get_feature_importance( data=Pool, type="FeatureImportance "PredictionValuesChange- "LossFunctionChange "ShapValues" )` — важности признаков и SHAP-значения.

- `eval_metrics(Pool, metrics=[...], plot=False)` — кривая метрик по итерациям на заданном `Pool`.
- `save_model("model.cbm")`, `load_model("model.cbm")` — сохранение/загрузка.
- `get_params()`, `set_params(**kwargs)` — доступ/изменение гиперпараметров.
- `tree_count_` — фактическое число построенных деревьев.

Что делают `eval_set`, `eval_metric`, `verbose` в `fit()`.

- `eval_set=[(X_tr,y_tr), (X_val,y_val)]` — набор датасетов, на которых *считаются* метрики на каждой итерации; на них не учатся, они для мониторинга и ранней остановки.
- `eval_metric` — метрика на `eval_set`. Для регрессии: `"rmse"`, `"mae"`, `"rmsle"`; для классификации: `"logloss"`, `"auc"`, `"error"` и т. п. Допускается список метрик.
- `verbose` — печать прогресса обучения: `True` печатает каждую итерацию (в sklearn-обёртке), `False` — молчание.

Код с ранней остановкой.

```
# pip install catboost
from catboost import CatBoostClassifier, Pool

cat_idx = [0, 3, 5]  # индексы категориальных колонок
train_pool = Pool(Xtr, ytr, cat_features=cat_idx)
valid_pool = Pool(Xval, yval, cat_features=cat_idx)

cat = CatBoostClassifier(
    iterations=5000, learning_rate=0.05, depth=8,
    l2_leaf_reg=5.0, loss_function="Logloss",
    eval_metric="AUC", random_state=7, verbose=200
)
cat.fit(train_pool, eval_set=valid_pool, use_best_model=True,
    early_stopping_rounds=200)
print("Best iters:", cat.get_best_iteration(), "Best score:",
    cat.get_best_score())
```

Тюнинг. Сначала подберите `depth` и `l2_leaf_reg`, затем уменьшайте `learning_rate` и увеличивайте `iterations` с ранней остановкой.

Пример: регрессия с CatBoost (`CatBoostRegressor`) с ранней остановкой.

```python
# pip install catboost
import numpy as np
from catboost import CatBoostRegressor, Pool

# Пример: X_tr, y_tr, X_val, y_val (y - вещественные)
# Если есть категориальные признаки, передайте их индексы в cat_features
cat_idx = [1, 3]  # пример: столбцы 1 и 3 - категориальные

train_pool = Pool(X_tr, y_tr, cat_features=cat_idx)
valid_pool = Pool(X_val, y_val, cat_features=cat_idx)

model = CatBoostRegressor(
    loss_function="RMSE",          # можно "MAE", "Quantile", "MAPE" и др.
    iterations=5000,               # верхняя граница числа деревьев
    learning_rate=0.05,            # шаг бустинга
    depth=8,                       # глубина симметричных деревьев
    l2_leaf_reg=6.0,               # L2-регуляризация на листьях
    bagging_temperature=0.8,       # сила стохастики (байесовский бутстрэп)
    rsm=0.9,                       # доля признаков на сплит (Random Subspace)
    eval_metric="RMSE",            # метрика валидации
    random_state=7,
    verbose=200
)

model.fit(
    train_pool,
    eval_set=valid_pool,
    use_best_model=True,           # сохранять лучшую по валидации
    early_stopping_rounds=300      # остановка при плато метрики
)

print("Best iter:", model.get_best_iteration())
print("Best score:", model.get_best_score())

# Предсказания на новых данных
y_pred = model.predict(X_test)

# Важности признаков (разные типы)
fi_gain = model.get_feature_importance(data=valid_pool,
    type="FeatureImportance")
fi_pvc  = model.get_feature_importance(data=valid_pool,
    type="PredictionValuesChange")
# SHAP-значения: массив (n_samples, n_features+1), последний столбец - базовое с
    мещение
shap    = model.get_feature_importance(data=valid_pool, type="ShapValues")
```

Краткие советы по тюнингу в регрессии.

- Сначала подберите `depth` (6–10) и `l2_leaf_reg` (3–10), затем уменьшайте `learning_rate` и повышайте `iterations` с *ранней остановкой*.
- При большом числе признаков используйте `rsm` < 1 и умеренный `bagging_temperature` для устойчивости.
- Для выбросов попробуйте `loss_function="MAE"` или `"Quantile"` (робастные альтернативы RMSE).

5.2.2 XGBoost

Сильные стороны: прозрачная регуляризация, гибкий контроль сложности.

Ключевые параметры.

- `n_estimators`, `learning_rate`, `max_depth`.
- `subsample`, `colsample_bytree` — стохастический бустинг.
- `min_child_weight` (минимальный вес в листе), `gamma` (штраф на сплит).
- `reg_lambda`, `reg_alpha` — L2/L1-регуляризация.
- `tree_method`: "hist"(быстро), "gpu_hist"(GPU).

Что означают `subsample`, `colsample_bytree`, `min_child_weight`.

- `subsample` (0..1) — доля объектов, случайно отбираемых для построения каждого дерева. Вносит стохастичность и снижает переобучение. Практика: 0.6–0.9; меньше — сильнее регуляризация, риск недообучения.
- `colsample_bytree` (0..1) — доля признаков, случайно отбираемых на уровне дерева (Random Subspace); уменьшает коррелированность деревьев и ускоряет обучение на высокоразмерных данных. Родственные: `colsample_bylevel`, `colsample_bynode`. Практика: 0.6–1.0.
- `min_child_weight` ≥ 0 — минимальная *сумма гессианов* в листе (аналог минимального «веса» листа). Больше $\Rightarrow$ сплиты агрессивнее отбрасываются, модель консервативнее (меньше мелких листьев). Для квадратичной ошибки близко к «минимальному числу объектов в листе». Практика: 1–5.

Код с early stopping.

```python
# pip install xgboost
from xgboost import XGBClassifier

xgb = XGBClassifier(
    n_estimators=5000, learning_rate=0.03, max_depth=6,
    subsample=0.8, colsample_bytree=0.8,
    min_child_weight=1.0, gamma=0.0,
    reg_lambda=1.0, reg_alpha=0.0,
    tree_method="hist", random_state=7, n_jobs=-1
)
xgb.fit(
    Xtr, ytr,
    eval_set=[(Xval, yval)], eval_metric="auc",
    verbose=200, # отображае результаты каждые 200 итераций
    early_stopping_rounds=300 # если в течение 300 итераций нет улучшений по мет
    рике, то останавливаемся (в любой момент)
)
print("Best n_estimators:", xgb.best_ntree_limit)
```

Тюнинг. Часто эффективно: уменьшить `learning_rate`, увеличить `n_estimators` с

`early_stopping`; контролировать сложность через `max_depth`, `min_child_weight`, `gamma`; добавить стохастику (`subsample`, `colsample_bytree`).

Пример: регрессия с ранней остановкой.

```python
# pip install xgboost
import numpy as np
from sklearn.model_selection import train_test_split
from xgboost import XGBRegressor

# Разбиение данных
X_tr, X_val, y_tr, y_val = train_test_split(X, y, test_size=0.2, random_state=7)

model = XGBRegressor(
    # базовые настройки бустинга
    n_estimators=5000,         # верхняя граница деревьев (early stopping останов
    ит раньше)
    learning_rate=0.03,        # шаг бустинга; меньше -> стабильнее, нужно больше
     деревьев
    max_depth=8,               # глубина деревьев

    # стохастичность и субсемплинг
    subsample=0.8,             # доля объектов на дерево
    colsample_bytree=0.8,      # доля признаков на дерево

    # регуляризация структуры
    min_child_weight=2.0,      # минимальный "вес" листа (гессиан-сумма)
    reg_lambda=1.0,            # L2-регуляризация (lambda)
    reg_alpha=0.0,             # L1-регуляризация (alpha)

    # производительность/детерминизм
    tree_method="hist",        # "hist" (CPU) или "gpu_hist" (GPU)
    random_state=7,
    n_jobs=-1
)

model.fit(
    X_tr, y_tr,
    eval_set=[(X_tr, y_tr), (X_val, y_val)],
    eval_metric="rmse",
    early_stopping_rounds=200,
    verbose=False
)

print("Best iteration:", model.best_iteration)
print("Best score (val rmse):", model.best_score)

y_pred = model.predict(X_test)
```

Небольшие советы по тюнингу.

- Стартовая сетка: `max_depth` $\in [6, 10]$, `min_child_weight` $\in [1, 5]$, `subsample` $\in [0.6, 0.9]$, `colsample_bytree` $\in [0.6, 1.0]$. Фиксируйте `learning_rate` (например, 0.05), ставьте большой `n_estimators` и включайте `early_stopping_rounds`.

- Если переобучает: увеличьте `min_child_weight`, уменьшите `max_depth`, понизьте `subsample`/`colsample_bytree`, усилите `reg_lambda`/`reg_alpha`, уменьшите `learning_rate`.
- Если недообучает: ослабьте регуляризацию, немного увеличьте `max_depth`/`subsample`/`colsample_`, либо увеличьте `n_estimators`.
- На больших числовых данных используйте `tree_method="hist"` (или `"gpu_hist"`) — существенно быстрее и экономичнее по памяти.

5.2.3 LightGBM

Сильные стороны: очень быстрый, leaf-wise рост, экономия памяти.

Ключевые параметры.

- `num_leaves` (главный рычаг сложности), `max_depth` (можно не задавать).
- `min_data_in_leaf`, `min_sum_hessian_in_leaf`.
- `feature_fraction`, `bagging_fraction`, `bagging_freq`.
- `lambda_l1`, `lambda_l2`, `min_gain_to_split`.
- `learning_rate`, `n_estimators`.

Что означают параметры.

- `min_data_in_leaf` (синоним: `min_child_samples`). Минимальное число объектов в листе. Увеличение делает модель более консервативной: меньше мелких листьев, ниже риск переобучения, но выше риск недообучения. Типичный диапазон: 20–200; на больших датасетах разумно увеличивать.
- `feature_fraction`. Доля признаков, случайно выбираемых для построения каждого дерева (аналог *colsample_ bytree*). Снижает корреляцию деревьев и ускоряет обучение. Типично 0.6–1.0. Меньше ⇒ сильнее регуляризация.
- `bagging_fraction`. Доля объектов, случайно выбираемых для каждого дерева (аналог *subsample*). Вносит стохастичность, уменьшает переобучение. Типично 0.6–0.9. Работает совместно с `bagging_freq`.
- `bagging_freq`. Как часто применять `bagging_fraction`: `0` — отключено; `k` > 0 — применять каждые k бустинговых итераций. Часто берут 1–5.
- `min_gain_to_split` (синоним: `min_split_gain`). Минимальное требуемое *увеличение* целевой функции для совершения сплита. Больше ⇒ реже сплиты, модель проще. Полезно, когда много шумовых признаков.

Код с ранней остановкой.

```python
# pip install lightgbm
from lightgbm import LGBMClassifier

lgbm = LGBMClassifier(
    n_estimators=5000, learning_rate=0.05,
    num_leaves=64, max_depth=-1, min_data_in_leaf=50,
    feature_fraction=0.8, bagging_fraction=0.8, bagging_freq=1,
    lambda_l1=0.0, lambda_l2=0.0, random_state=7, n_jobs=-1
)
lgbm.fit(
    Xtr, ytr,
    eval_set=[(Xval, yval)], eval_metric="auc",
    callbacks=[lambda env: None],  # чтобы быть совместимым с старым/новым API
    verbose=200
)
# В новых версиях: early_stopping(300) через callbacks
```

Тюнинг. Согласуйте `num_leaves` и `min_data_in_leaf`; снижайте `learning_rate` и повышайте `n_estimators` с ранней остановкой; включайте `feature_fraction`/`bagging_fraction`.

Пример: регрессия с ранней остановкой.

```python
# pip install lightgbm
import numpy as np
from sklearn.model_selection import train_test_split
from lightgbm import LGBMRegressor

# Разбиение на train/val
X_tr, X_val, y_tr, y_val = train_test_split(X, y, test_size=0.2, random_state=7)

model = LGBMRegressor(
    # базовые настройки бустинга
    n_estimators=5000,          # верхняя граница деревьев (early stopping останов
    learning_rate=0.03,         # шаг бустинга: меньше -> стабильнее, нужно больше
    max_depth=-1,               # без жёсткого ограничения глубины (контроль через
    num_leaves=64,              # сложность дерева; ~ 2^(глубина)

    # стохастичность по объектам и признакам
    feature_fraction=0.8,       # доля признаков на дерево (colsample)
    bagging_fraction=0.8,       # доля объектов на дерево (subsample)
    bagging_freq=1,             # как часто применять bagging (каждую итерацию)

    # регуляризация структуры
    min_data_in_leaf=80,        # минимальное число объектов в листе
    min_gain_to_split=0.0,      # минимальный прирост для сплита (можно >0 для упр

    # регуляризация листов/весов
    reg_lambda=1.0,             # L2-регуляризация
    reg_alpha=0.0,              # L1-регуляризация

    # производительность/детерминизм
    random_state=7,
    n_jobs=-1
)

# обучаем с мониторингом валидации и ранней остановкой
model.fit(
    X_tr, y_tr,
    eval_set=[(X_tr, y_tr), (X_val, y_val)],
    eval_metric="rmse",
    callbacks=[  # эквивалент early_stopping_rounds в sklearn-обёртке < 4.0
        # lightgbm.early_stopping(stopping_rounds=200, verbose=False),
        # lightgbm.log_evaluation(period=0)
    ]
)

# Предсказания и полезные атрибуты
y_pred = model.predict(X_test, num_iteration=model.best_iteration_)
# Важности признаков:
importances = model.feature_importances_
```

Примечания.

- Если установлен `lightgbm` $\geq$`4.x`, вместо аргумента `early_stopping_rounds` используйте колбэк `lightgbm.early_stopping`. В версиях $<$ `4.0` можно передавать `early_stopping_rou`

прямо в `fit()`.

- `feature_fraction` и `bagging_fraction` работают только если `bagging_freq` > 0.
- Для категориальных признаков LightGBM поддерживает нативный сплит: передайте `categorical_feature` или индексы колонок (лучше `pandas` с `category` dtype).

Небольшие советы по тюнингу.

- Начните с: `num_leaves` $\approx 2^{\texttt{max\_depth}}$, `min_data_in_leaf` $= 50$–100, `feature_fraction` $= 0.8$, `bagging_fraction` $= 0.8$, `bagging_freq` $= 1$, `learning_rate` $= 0.05$, большой `n_estimators` с ранней остановкой.
- Если видите переобучение: увеличьте `min_data_in_leaf`, `min_gain_to_split`, уменьшите `num_leaves`, `feature_fraction` и `bagging_fraction`.
- Если недообучает: уменьшите `min_data_in_leaf` и `min_gain_to_split`, слегка увеличьте `num_leaves` и фракции.

Пример `RandomizedSearchCV` (XGBoost).

```python
from sklearn.model_selection import RandomizedSearchCV
from xgboost import XGBClassifier
import numpy as np

param_dist = {
    "max_depth": [4, 6, 8, 10],
    "learning_rate": np.logspace(-2, -0.3, 8),
    "subsample": [0.6, 0.8, 1.0],
    "colsample_bytree": [0.6, 0.8, 1.0],
    "min_child_weight": [1, 3, 5],
    "reg_lambda": [0.0, 0.5, 1.0, 2.0]
}
base = XGBClassifier(n_estimators=4000, tree_method="hist", n_jobs=-1,
    random_state=7)
rs = RandomizedSearchCV(
    base, param_dist, n_iter=30, scoring="roc_auc", cv=5, n_jobs=-1,
    random_state=7
).fit(Xtr, ytr)
print(rs.best_params_, rs.best_score_)
```

Идея. `RandomizedSearchCV` случайно выбирает `n_iter` сочетаний гиперпараметров из `param_distributions`, для каждого сочетания проводит кросс-валидацию `cv` по метрике `scoring`, находит лучшие параметры и (по умолчанию) переобучает модель на всём трейне. (Код написан для двуклассовой классификации, так как использует `roc_auc`. Для многоклассовой классификации можно выбрать следующие метрики: `accurancy`, `f1`$_{macro}$/`f1`$_{weighted}$($macro-$, $weighted-$), `roc_auc_ovr`, `roc_auc_ovo`.

```python
from sklearn.model_selection import RandomizedSearchCV

rs = RandomizedSearchCV(
    estimator=base,                  # любая модель или Pipeline
    param_distributions=param_dist, # словарь или распределения (scipy.stats)
    n_iter=30,                       # сколько случайных наборов попробовать
    scoring="roc_auc",               # метрика качества
    cv=5,                            # CV-стратегия (число фолдов или объект CV)
    n_jobs=-1,                       # параллелизм: все ядра
    random_state=7,                  # воспроизводимость сэмплинга
    refit=True,                      # после поиска переобучить на всём трейне
    return_train_score=False,        # при необходимости вернуть train-оценки
)
# rs.fit(X, y)  ->  rs.best_params_, rs.best_score_, rs.best_estimator_,
    rs.cv_results_
```

Разбор ключевых аргументов.

- estimator (base) - базовая модель (например, Pipeline([... , ("clf LogisticRegressio
- param_distributions (param_dist) - откуда сэмплировать гиперпараметры:
  - дискретные списки: {"clf__C": [0.1, 1, 10]};
  - непрерывные распределения scipy.stats: {"clf__C": loguniform(1e-3, 1e3)};
  - имена параметров в Pipeline: <шаг>__<параметр>.
- n_iter - число случайных комбинаций. Больше - дольше, но шанс найти лучше выше.
- scoring - строка/функция метрики ("roc_auc", "average_precision", "neg_log_loss", "accuracy", "r2", "neg_mean_absolute_error и т.д.).
- cv - число фолдов или объект сплита (StratifiedKFold, GroupKFold, TimeSeriesSplit).
- n_jobs - параллельные процессы (-1 - все).
- random_state - фиксирует сэмплинг из param_distributions.
- refit - если True, переобучает best_estimator_ на всём X, y.
- return_train_score - вернуть/не вернуть train-метрики в cv_results_.

Что происходит при fit(X, y).

1. n_iter раз сэмплируются гиперпараметры из param_distributions.
2. Для каждого набора выполняется cv-разбиение, модель обучается на трейне каждого фолда и оценивается на валидации по scoring.
3. Считается средняя валидационная оценка по фолдам; выбирается лучший набор.
4. Если refit=True, модель с best_params_ переобучается на всём трейне; доступны best_estimator_, best_score_, best_params_, cv_results_.

Полезные советы.

- Заворачивайте препроцессинг в Pipeline, чтобы всё обучалось *внутри* фолдов и не было утечек.
- Для несбалансированных классов используйте scoring="roc_auc" или "average_precision для многокласса - "roc_auc_ovr" / "roc_auc_ovo" (_weighted, если классы неравновесн

- Если есть группы (один пользователь в разных сэмплах) - cv=GroupKFold(...) и rs.fit(X, y, groups=groups).

5.3 Блендинг (blending)

Идея. Делим обучающую выборку на *train* и *holdout*. Базовые модели учим на *train*, получаем их предсказания на *holdout*, обучаем простую мета-модель на этих признаках-пред

Мини-код (бинарная классификация).

```
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import roc_auc_score

X_tr, X_hold, y_tr, y_hold = train_test_split(X, y, test_size=0.2, stratify=y,
    random_state=7)

m1 = RandomForestClassifier(n_estimators=600, n_jobs=-1,
    random_state=7).fit(X_tr, y_tr)
m2 = XGBClassifier(n_estimators=2000, learning_rate=0.05, tree_method="hist",
                   n_jobs=-1, random_state=7).fit(X_tr, y_tr)

P1_hold = m1.predict_proba(X_hold)[:, 1]
P2_hold = m2.predict_proba(X_hold)[:, 1]
# склеивание массивов по колонкам
Z_hold = np.c_[P1_hold, P2_hold]

meta = LogisticRegression(max_iter=200).fit(Z_hold, y_hold)

# Инференс на новом X_new: сначала базовые, затем мета
P1_new = m1.predict_proba(X_new)[:, 1]
P2_new = m2.predict_proba(X_new)[:, 1]
P_final = meta.predict_proba(np.c_[P1_new, P2_new])[:, 1]

print("Blend AUC:", roc_auc_score(y_hold, meta.predict_proba(Z_hold)[:,1]))
```

Замечание. Блендинг использует только один holdout; менее «бережлив» к данным, чем стэкинг с KFold.

5.4 Стэкинг (stacking)

Идея. Кросс-валидируем базовые модели: для каждого фолда получаем OOF-предсказания (out-of-fold). Эти OOF составляют обучающую матрицу для мета-модели. На инференсе мета-модель берёт средние/конкатенированные предсказания базовых, обученных на всём трейне.

Код на sklearn.

```python
from sklearn.ensemble import StackingClassifier, StackingRegressor
from sklearn.svm import SVC
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler

estimators = [
    ("rf", RandomForestClassifier(n_estimators=500, n_jobs=-1, random_state=7)),
    ("xgb", XGBClassifier(n_estimators=2000, learning_rate=0.05,
    tree_method="hist",
                          n_jobs=-1, random_state=7)),
    ("svm", make_pipeline(StandardScaler(with_mean=False),  # если разреженно
                          SVC(probability=True, kernel="rbf", C=2.0)))
]

stack = StackingClassifier(
    estimators=estimators,
    final_estimator=LogisticRegression(max_iter=300),
    cv=5, n_jobs=-1, passthrough=False
)

print("Stack AUC:", cross_val_score(stack, Xtr, ytr, cv=5, scoring="roc_auc",
    n_jobs=-1).mean())
```

Советы. Старайтесь, чтобы базовые модели были «разнотипными» (разные inductive biases). Для разреженных матриц используйте линейные модели/ядровые с нормализацией. Обязательно отделяйте валидацию (не допускайте утечек между базовыми и мета-моделью).

5.5 StackingClassifier и StackingRegressor: параметры и примеры

StackingClassifier: основные параметры.

- estimators: список пар (имя, модель). Любая модель sklearn или Pipeline. Можно указать "drop" вместо модели, чтобы временно исключить базовый алгоритм.
- final_estimator: мета-модель (по умолчанию LogisticRegression()). Обучается на OOF-признаках (предсказаниях базовых моделей с кросс-валидации).
- stack_method: как превращать предсказания базовых моделей в мета-признаки. Допустимые значения: "auto" (приоритет predict_proba $\rightarrow$ decision_function $\rightarrow$ predict), "predict_proba", "decision_function", "predict".
- passthrough (False по умолчанию): если True, к мета-признакам добавляются исходные признаки X.
- cv: число фолдов или объект сплита (StratifiedKFold, GroupKFold, TimeSeriesSplit и т. д.). Управляет получением OOF-предсказаний, предотвращая утечки.
- n_jobs: параллелизм при кросс-валидации базовых моделей (-1 - все ядра).
- verbose: уровень логов (целое, по умолчанию 0).

Атрибуты.

- named_estimators_: словарь обученных базовых моделей.
- final_estimator_: обученная мета-модель.
- classes_ (для классификации), n_features_in_, feature_names_in_ (если доступны).

Пример: StackingClassifier **(вероятности для мета-модели).**

```python
from sklearn.ensemble import StackingClassifier, RandomForestClassifier
from sklearn.svm import SVC
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import StratifiedKFold, cross_val_score

base_estimators = [
    ("lr",  make_pipeline(StandardScaler(), LogisticRegression(max_iter=200))),
    ("svc", make_pipeline(StandardScaler(), SVC(probability=True))),
    ("rf",  RandomForestClassifier(n_estimators=300, random_state=7))
]

meta = LogisticRegression(max_iter=300, class_weight="balanced")

cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=7)

clf = StackingClassifier(
    estimators=base_estimators,
    final_estimator=meta,
    stack_method="predict_proba",    # явно берём вероятности
    passthrough=False,               # только мета-признаки от базовых моделей
    cv=cv,
    n_jobs=-1,
    verbose=0
)

# clf.fit(X_train, y_train); y_proba = clf.predict_proba(X_test)[:, 1]
# Оценка по CV:
# scores = cross_val_score(clf, X, y, scoring="roc_auc", cv=cv, n_jobs=-1)
```

Замечания по практике.

- Если мета-модель принимает вероятности, имеет смысл принудить stack_method="predict
  и обеспечить probability=True у SVC.
- Для линейных моделей в базовом слое полезно масштабирование (StandardScaler
  внутри Pipeline).
- passthrough=True может помочь, если исходные признаки содержат важную информацию,
  которую базовые модели «сгладили».

StackingRegressor: параметры (отличия).

- estimators, final_estimator, passthrough, cv, n_jobs, verbose - аналогично
  классификатору.
- Параметра stack_method нет: для регрессии всегда используется predict.
- По умолчанию final_estimator=Ridge() (в разных версиях sklearn - LinearRegression()
  мета-модель можно заменить на Lasso/ElasticNet/XGBRegressor/LightGBM и т. д.

Пример: StackingRegressor.

```python
from sklearn.ensemble import StackingRegressor, RandomForestRegressor,
    GradientBoostingRegressor
from sklearn.linear_model import Ridge, ElasticNet
from sklearn.model_selection import KFold, cross_val_score
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler

reg_estimators = [
    ("ridge", make_pipeline(StandardScaler(), Ridge(alpha=1.0))),
    ("gbr",   GradientBoostingRegressor(random_state=7)),
    ("rf",    RandomForestRegressor(n_estimators=400, random_state=7))
]

meta_reg = ElasticNet(alpha=0.01, l1_ratio=0.2, random_state=7)

cv = KFold(n_splits=5, shuffle=True, random_state=7)

reg = StackingRegressor(
    estimators=reg_estimators,
    final_estimator=meta_reg,
    passthrough=True,     # добавим исходные признаки к мета-признакам
    cv=cv,
    n_jobs=-1,
    verbose=0
)

# reg.fit(X_train, y_train); y_pred = reg.predict(X_test)
# Оценка по CV (MAE со знаком минус в sklearn):
# scores = cross_val_score(reg, X, y, scoring="neg_mean_absolute_error", cv=cv,
    n_jobs=-1)
# mae = -scores.mean()
```

Когда стэкинг особенно полезен.

- Базовые модели дополняют друг друга (разная природа ошибок: линейные + деревья + ядра).
- Мета-модель получает достаточно «богатые» признаки (вероятности/скор) и хорошо регуляризуется.
- Правильно настроенный cv исключает утечки; с группами используйте GroupKFold и передавайте groups в fit.

6 Почему важны распределения и зачем нормализовать данные

1) Зачем нужны распределения.

- Описание неопределённости. Распределение моделирует, как «случайно» возникают наблюдения: шум измерений, вариабельность клиентов, длительности событий и т.п.
- Вероятностные выводы. Через распределения считаем вероятности, квантиили, доверител интервалы, p-значения и правдоподобие (лог-ликелихуд) для обучения.
- Выбор потерь и моделей. Нормальное $\Rightarrow$ MSE/линейная регрессия; Бернулли $\Rightarrow$ логистическая; Пуассон $\Rightarrow$ счётные данные; Экспоненциальное/Вейбулла $\Rightarrow$ времена жизни.
- Синтетика и симуляции. Сэмплирование из известных законов для тестирования алгоритмов и стресс-тестов.
- Регуляризация и априоры. В байесовских методах распределения задают априорные знания (напр., Гаусс для весов, Лаплас для разреженности).

2) Зачем и когда нормализовать/стандартизовать.

- Стандартизация (Z-score): вычесть среднее и поделить на СКО по признаку $\Rightarrow$ ноль среднее, единичная дисперсия. Нужна там, где *масштаб признаков влияет*:
  - методы на расстояниях/скалярных произведениях: KNN, SVM (RBF/linear), kMeans, PCA/TSNE/UMAP;
  - градиентные модели: Logistic/Linear Regression (с регуляризацией), нейросети (ускоряет схождение);
  - модели чувствительные к регуляризации (C, alpha, weight_decay) - единый масштаб делает штраф «честным».
- Нормализация (мин-макс к $[0, 1]$ или приведение *векторной нормы* строки к 1) - полезна для интерпретации, визуализаций, нейросетей с сигмоидой/тангенсом, KNN/cosine метрик.
- Робастное масштабирование (к медиане и IQR) - устойчиво к выбросам.
- Когда можно *не* масштабировать. Деревья, случайный лес, бустинги (CatBoost/XGBoost/L почти не чувствительны к монотонным преобразованиям масштаба.
- Важное правило против утечек. *Всегда* подгоняйте скейлер fit *только на трейне*, применяйте transform к валидации/тесту. Удобнее всего - через Pipeline.

3) Как стандартизировать и нормализовать данные (Python)

```python
# --- Стандартизация (Z-score) с защитой от утечек через Pipeline ---
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import cross_val_score, StratifiedKFold

pipe = make_pipeline(
    StandardScaler(),                       # fit только на трейне каждого фолда
    LogisticRegression(solver="lbfgs", max_iter=200)
)

cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
scores = cross_val_score(pipe, X, y, scoring="roc_auc", cv=cv, n_jobs=-1)

# --- Мин-макс нормализация в [0, 1] ---
from sklearn.preprocessing import MinMaxScaler
mm = MinMaxScaler()          # mm.fit(X_train); Xtr = mm.transform(X_train); Xte =
    mm.transform(X_test)

# --- Робастный скейлер (к медиане и IQR), полезен при выбросах ---
from sklearn.preprocessing import RobustScaler
rb = RobustScaler()          # rb.fit(X_train); ...

# --- Нормализация векторной нормы строк (L2 по строкам) ---
from sklearn.preprocessing import Normalizer
norm = Normalizer(norm="l2")  # превращает каждую строку x -> x / ||x||_2; полезно д
    ля cosine, KNN

# --- Правильно смешиваем препроцессинг числовых/категориальных столбцов ---
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import OneHotEncoder

num_cols = ["age","income"]
cat_cols = ["city","segment"]

pre = ColumnTransformer(
    transformers=[
        ("num", StandardScaler(), num_cols),
        ("cat", OneHotEncoder(handle_unknown="ignore"), cat_cols)
    ],
    remainder="drop"
)

from sklearn.svm import SVC
pipe2 = make_pipeline(pre, SVC(kernel="rbf", probability=True))
# pipe2.fit(X_train_df, y_train); pipe2.predict_proba(X_test_df)

# --- Ручной Z-score (важно: среднее/СКО считаем только на трейне!) ---
import numpy as np
mu, sigma = X_train.mean(axis=0), X_train.std(axis=0, ddof=0)
Xtr = (X_train - mu) / (sigma + 1e-12)
Xte = (X_test  - mu) / (sigma + 1e-12)
```

Итого. Распределения дают язык для неопределённости, вероятностных выводов и выбора адекватных потерь/моделей. Масштабирование признаков - практический must-have для расстояний, ядровых методов, PCA и градиентных моделей. Делайте fit скейлеров *только* на трейне (через Pipeline/ColumnTransformer) и подбирайте тип скейлинга: StandardScaler (по умолчанию), MinMaxScaler (для ограниченного диапазона), RobustScaler (при выбросах), Normalizer (косинус/текст/TF-IDF).