Практическая справка по тому, как писать код

Содержание

1 Оценка качества: кросс-валидация и бутстрэп **3**

1.1 Общие правила против утечек данных 3

1.2 K-Fold кросс-валидация (классификация с масштабированием) 3

1.3 Параметры `cross_val_score` (кратко) 4

1.4 Интерпретация вывода 4

1.5 Leave-One-Out (LOO) — максимально строгая CV 4

1.6 Leave-One-Group-Out (LOGO) — запрет утечек между группами 5

1.7 Бутстрэп-оценка и доверительные интервалы метрики 5

1.8 Подбор гиперпараметров без утечек (намёк) 6

1.9 Чеклист «без утечек» 6

2 Оптимизация гиперпараметров: практическая выжимка по ноутбуку в PyTorch **8**

2.1 Несколько важных понятий перед разбором ноутбука 8

2.2 torch.optim.LBFGS 12

2.3 torch.optim.Adam 15

2.4 Финальный расчёт меток и центроидов (инференс без градиентов). 17

2.5 Метрика силуэта (`silhouette_score`). 18

2.6 Что возвращает функция (словарь результатов). 20

2.7 Короткие практические выводы 21

2.8 Мини-пример: настройка регуляризации и оптимизация модели (PyTorch, Adam). 22

2.8.1 Контейнер `nn.Sequential`: что это, что внутрь кладут, есть ли свои параметры . . . 23

2.8.2 Что такое логиты (*logits*) . . . 23

2.8.3 `nn.Linear`: что это, как работает и зачем нужен . . . 24

2.8.4 `nn.BCEWithLogitsLoss`: что это и как работает . . . 26

3 Оптимизаторы `sklearn`: `SGDClassifier` и `SGDRegressor` 32

3.1 `SGDClassifier`: линейный классификатор . . . 32

3.2 Допустимые значения параметров: `loss` и `elasticnet` . . . 32

3.3 `SGDRegressor`: линейная регрессия с SGD . . . 34

3.4 Функции потерь в `SGDRegressor`: `huber`, `epsilon_insensitive`, `squared_epsilon_insensiti`

3.5 Практика и советы . . . 36

3.6 Ещё короткие примеры . . . 36

3.7 Параметр `solver` в scikit-learn: где он есть и что означает . . . 37

4 KNN: классифицирующие и регрессионные варианты 39

4.1 Взвешенный KNN: `weights="distance"` и свои ядра . . . 40

4.2 K-Means (`sklearn.cluster.KMeans`) . . . 42

5 Решающие деревья в `scikit-learn`: обучение, стрижка, кодирование категорий 45

5.1 Бинарное решающее дерево (классификация и регрессия) . . . 45

5.2 Стрижка дерева (Pruning): cost-complexity и выбор `ccp_alpha` . . . 46

5.3 Категориальные признаки: label/one-hot/target encoding . . . 49

6 SVM и SVR: быстрый практический конспект 53

6.1 SVM для классификации (`SVC`, `LinearSVC`) . . . 53

6.2 SVR для регрессии (`SVR`, `LinearSVR`) . . . 54

1 Оценка качества: кросс-валидация и бутстрэп

Зачем. Надёжная оценка обобщающей способности модели требует *строгого разделения* данных на обучающие и валидационные части без «подглядывания» в ответы валидации. Ниже — практические шаблоны кода и правила, как не допустить утечек.

1.1 Общие правила против утечек данных

- **Весь препроцессинг — внутри `Pipeline`.** Масштабирование, целевое кодирование, отбор признаков и т.п. должны обучаться ТОЛЬКО на тренировочной части каждого фолда.
- **Не трогайте тест.** Любой подбор гиперпараметров делайте *только* по CV на трейне. Тест держите «на потом».
- **Соблюдайте стратификацию/группы.** Для классификации используйте `StratifiedKFold`. Для зависимых выборок (пользователь, серия, сессия) используйте `GroupKFold`/`LeaveOneGroupOu`
- **Временные ряды — отдельный режим.** Для них применяют `TimeSeriesSplit` (разрезы «вперёд во времени»).

1.2 K-Fold кросс-валидация (классификация с масштабированием)

```python
from sklearn.model_selection import StratifiedKFold, cross_val_score
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression

cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
pipe = make_pipeline(
    StandardScaler(),
    LogisticRegression(solver="lbfgs", max_iter=200)
)

scores = cross_val_score(
    pipe, X, y,
    scoring="roc_auc",
    cv=cv,
    n_jobs=-1
)
print("AUC (meanstd): %.3f  %.3f" % (scores.mean(), scores.std()))
```

Комментарий. `StandardScaler` обучается *внутри каждого фолда*, исключая утечки масштаба. Для регрессии используйте `KFold` вместо `StratifiedKFold` и подходящую метрику (`r2`, `neg_mean_absolute_error`, ...).

Что происходит.

- Генератор `StratifiedKFold` создаёт 5 стратифицированных фолдов (сохраняются доли классов), перемешивание фиксируется `random_state`.
- `Pipeline` гарантирует, что `StandardScaler` обучается *внутри каждого фолда* только на его train-части, исключая утечку масштаба.

- `cross_val_score` для каждого фолда: клонирует пайплайн, обучает на train, считает метрику ROC AUC на val; возвращает массив из 5 значений.
- `n_jobs=-1` параллелит вычисления по всем доступным ядрам.

1.3 Параметры `cross_val_score` (кратко)

- `estimator`: любая модель/пайплайн с методами `fit` и `predict`/`predict_proba`/`decision_functi`
- `X, y`: данные и целевая переменная; для безнадзорных задач `y` можно опустить.
- `groups`: групповые метки (используются, если `cv` — групповой сплит).
- `scoring`: строка с метрикой (`"roc_auc"`, `"accuracy"`, `"neg_mean_absolute_error` и др.) либо кастомная функция-оценщик.
- `cv`: число фолдов (например, 5) или генератор разбиений (`StratifiedKFold`, `KFold`, `GroupKFold`, `TimeSeriesSplit`).
- `n_jobs`: число параллельных процессов; `-1` — все ядра.
- `verbose`, `pre_dispatch`, `error_score`: управление логами, диспетчеризацией задач и поведением при ошибках (по умолчанию `np.nan`).

Возвращаемое значение: `ndarray` формы $(n\_splits,)$ — метрика на каждом фолде. Если нужны несколько метрик и/или времена `fit`/`score`, используйте `cross_validate`.

1.4 Интерпретация вывода

Строка печати `"AUC (mean+/-std): ..."` показывает среднее качество по фолдам и его разброс. Небольшая дисперсия (std мала) — признак стабильности; большая — сигнал проверить стратификацию, несбалансированность классов и/или корректность препроцессинга внутри `Pipeline`.

1.5 Leave-One-Out (LOO) — максимально строгая CV

```
from sklearn.model_selection import LeaveOneOut, cross_val_score
from sklearn.linear_model import SGDClassifier
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler

loo = LeaveOneOut()
pipe = make_pipeline(StandardScaler(), SGDClassifier(loss="log_loss",
    random_state=7))
scores = cross_val_score(pipe, X, y, scoring="accuracy", cv=loo)
print("LOO accuracy (mean):", scores.mean())
```

Комментарий. LOO делает N прогонов (дорого), каждый раз обучая на $N-1$ примерах и проверяя на 1 примере. Полезно на маленьких датасетах; на средних/больших — неэффективно.

1.6 Leave-One-Group-Out (LOGO) — запрет утечек между группами

```python
from sklearn.model_selection import LeaveOneGroupOut, cross_val_score
from sklearn.neighbors import KNeighborsClassifier
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
import numpy as np

# groups[i] -- идентификатор группы (например, пользователь/сессия/серия)
groups = np.array(user_ids)  # длины N

logo = LeaveOneGroupOut()
pipe = make_pipeline(StandardScaler(), KNeighborsClassifier(n_neighbors=5))

scores = cross_val_score(
    pipe, X, y,
    groups=groups,      # <- критично: разрезы делаются по группам
    scoring="accuracy",
    cv=logo
)
print("LOGO accuracy (meanstd): %.3f  %.3f" % (scores.mean(), scores.std()))
```

Комментарий. Все объекты одной группы *целиком* попадают либо в трейн, либо в валидацию — это устраняет утечки (например, когда один и тот же пользователь встречается в обоих наборах).

1.7 Бутстрэп-оценка и доверительные интервалы метрики

Идея. С переотбором (*bootstrap*) многократно семплируем тренировочные выборки из исходных данных, обучаем модель и считаем метрику на *out-of-bag* (примеров, не попавших в бутстрэп-выборку) или на фиксированном валидационном наборе. Распределение метрики даёт стандартную ошибку и доверительный интервал.

```python
import numpy as np
from sklearn.utils import resample
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression

# фиксируем тест, бутстрэпим трейн (чтобы не "подглядывать" в тест)
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.2, stratify=y,
    random_state=7)

pipe = make_pipeline(StandardScaler(), LogisticRegression(max_iter=1000))
B = 1000
scores = []

rng = np.random.RandomState(7)
n = len(y_tr)

for b in range(B):
    idx = rng.randint(0, n, size=n)        # семпл с возвращением
    Xb, yb = X_tr[idx], y_tr[idx]
    pipe.fit(Xb, yb)
    # оценка на фиксированном тесте (без утечки)
    scores.append( accuracy_score(y_te, pipe.predict(X_te)) )

scores = np.array(scores)
lo, hi = np.percentile(scores, [2.5, 97.5])
print("Bootstrap acc: mean=%.3f  95%% CI=[%.3f, %.3f]" % (scores.mean(), lo,
    hi))
```

Комментарий. Альтернатива — оценивать на *out-of-bag* каждом прогоне: брать объекты, *не* вошедшие в бутстрэп-выборку, и считать метрику по ним. Важно: *не* использовать всю исходную разметку для настройки препроцессинга — держите препроцесс в `Pipeline` и обучайте его только на бутстрэп-выборке.

1.8 Подбор гиперпараметров без утечек (намёк)

Правильный контур.

- Для подбора `C`, `alpha`, `n_neighbors`, ... используйте `GridSearchCV`/`RandomizedSearchCV` с `Pipeline` и корректным `cv` (`StratifiedKFold`, `GroupKFold`, ...).
- **Нестед CV** (вложенная CV) даёт незамещённую оценку качества, но дороже по вычислениям.

1.9 Чеклист «без утечек»

- Все трансформеры/кодировщики/скалеры — внутри `Pipeline`.
- Группы/стратификация указаны явно, если это важно для задачи.
- Гиперпараметры подбираются *внутри* CV, тест используется один раз — в самом конце.

- Для временных рядов — `TimeSeriesSplit`, а не обычный KFold.

2 Оптимизация гиперпараметров: практическая выжимка по ноутбуку в PyTorch

В ноутбуке решаются две задачи: мягкая кластеризация k-means на наборах `Iris`. Для кластеризации оптимизируются координаты центроидов с помощью двух оптимизаторов (`LBFGS`, `Adam`); в классификации сравниваются `LogisticRegression` (L-BFGS) и `AdamW` в `PyTorch`. Качество в кластеризации оценивается метрикой `silhouette_score`. Ниже — подробные пояснения ко всем вызовам.

2.1 Несколько важных понятий перед разбором ноутбука

Центроиды — это ключевое понятие в алгоритме k-means. Под центроидом понимается «центр» кластера: точка в пространстве признаков, которая играет роль представителя группы объектов. Если у нас есть k кластеров, то мы поддерживаем k центроидов, обычно в виде матрицы размера $k \times d$, где d — размерность признакового пространства. Каждая строка такой матрицы хранит координаты одного центроида. В классическом k-means на каждом шаге объекты относятся к ближайшему центроиду, после чего координаты центроидов пересчитываются как среднее по своим точкам. В мягкой (soft) версии k-means, которая реализована в ноутбуке, назначение точек к центроидам происходит не жёстко, а через распределение вероятностей (softmax от расстояний). Центроиды при этом становятся *параметрами* оптимизации: мы ищем такие их значения, которые минимизируют функцию потерь, отражающую «качество» разбиения. Именно эти центроиды и передаются в оптимизатор (`LBFGS`, `Adam`) как обучаемый параметр `C`.

Выбор начальных центроидов — важный шаг в алгоритме k-means. Так как задача кластеризации нелинейна и может иметь много локальных минимумов, начальная инициализация сильно влияет на итоговый результат. В классическом варианте k-means центроиды могут быть выбраны просто случайным образом из множества объектов: берём k случайных точек и используем их координаты как начальные центры. Более продвинутый способ — инициализация `k-means++`, где новые центры выбираются так, чтобы они были как можно дальше друг от друга; это улучшает устойчивость и снижает риск попадания в плохой минимум. В нашем ноутбуке для простоты использовалась случайная инициализация: матрица `init_centroids` формируется либо из случайного подмножества данных, либо из случайных чисел, после чего она превращается в параметр `nn.Parameter` и дальше уже оптимизируется выбранным методом (`LBFGS`, `SGD`, `Adam`). Таким образом, оптимизация не «придумывает» кластеры с нуля, а постепенно улучшает первоначальную случайную расстановку центроидов.

Чтобы понимать, что именно мы оптимизируем, важно уточнить несколько базовых понятий. В PyTorch все данные представлены в виде *тензоров*. Тензор — это многомерный массив чисел: вектор (одномерный тензор), матрица (двумерный тензор), трёхмерный массив (например, для изображений формата `[каналы, высота, ширина]`), и так далее. В задаче кластеризации у нас есть матрица признаков $X \in \mathbb{R}^{N \times d}$, где N — число объектов, d — размерность признакового пространства. Аналогично, начальные центроиды можно хранить как матрицу размера $k \times d$, где k — количество кластеров, а каждая строка соответствует координатам одного центра. Такой объект в PyTorch — это обычный тензор, созданный заранее (например, из случайной инициализации или через выбор случайных объектов из данных).

Когда мы пишем `nn.Parameter(init_centroids.clone().to(device), requires_grad=True)`, мы именно этот тензор центроидов превращаем в обучаемый параметр. Мы сами явно задаём его (через переменную `init_centroids`), и именно его оптимизатор будет изменять. Таким образом, тензоры *не появляются «из воздуха»* — их создаёт исследователь, а класс `nn.Parameter` лишь «оборачивает» их, чтобы PyTorch учитывал их в процессе оптимизации

В этом разделе на простом примере разбирается то, как использовать оптимизаторы `LBFGS` и `Adam` для оптимизации параметров. Также разбирается, что такое «температура» `tau` в мягком k-means и, как работает `soft_`k`-means_loss` функция с использованием `softmax`.

Что делает `soft_kmeans_loss` и почему она так устроена.
Это скалярная функция потерь, которую мы минимизируем по параметрам (центроидам) для задачи "мягкой" кластеризации. На вход она получает батч данных $X_{\text{batch}} \in \mathbb{R}^{B\times D}$ и текущие центроиды $C \in \mathbb{R}^{K\times D}$; на выходе возвращает одно число L, измеряющее, насколько хорошо центры объясняют точки из батча.

```
def soft_kmeans_loss(centroids: torch.Tensor,
    X_batch: torch.Tensor,
    tau: float = 0.5) -> torch.Tensor:

    # X_batch: (B, D), centroids: (K, D)
    # 1) Попарные квадраты расстояний d_{b,k} = ||x_b - c_k||^2  ->  (B, K)
    diff = X_batch[:, None, :] - centroids[None, :, :]
    dist2 = (diff * diff).sum(dim=2)

    # 2) Мягкие присваивания по softmax с температурой tau
    q = torch.softmax(-dist2 / tau, dim=1)  # (B, K)

    # 3) Ожидаемое расстояние (реконструкция)
    recon = (q * dist2).mean()

    # 4) Энтропийный член (поощряет мягкость на ранних этапах)
    eps = 1e-12
    entropy = -(q * torch.log(q + eps)).mean()

    # 5) Итоговый лосс
    loss = recon - tau * entropy
    return loss
```

Сначала считаются попарные квадраты расстояний от каждой точки до каждого центроида. За счёт броадкастинга формируется тензор разностей `diff` $\in \mathbb{R}^{B\times K\times D}$ и затем `dist2` $= (\texttt{diff}\cdot\texttt{diff})$ суммируется по признаковой оси: `dist2` $\in \mathbb{R}^{B\times K}$ с элементами $d_{b,k} = \|x_b - c_k\|_2^2 = \sum_{d=1}^{D}(x_{b,d} - c_{k,d})^2$. Это «геометрическая часть» лосса.

Почему появляется трёхмерный тензор $(B \times K \times D)$ и как в этом участвует broadcasting.
Нам нужно сразу получить расстояния от *каждой* точки батча к *каждому* центроиду. Пусть $X_{\text{batch}} \in \mathbb{R}^{B\times D}$ — это B точек в D-мерном пространстве, а $C \in \mathbb{R}^{K\times D}$ — K

центроидов. Попарные разности координат для всех пар (b, k) удобно хранить в едином тензоре `diff` с формой (B, K, D), где элемент

$$\texttt{diff}[b, k, d] \;=\; X_{\text{batch}}[b, d] \;-\; C[k, d].$$

Такой трёхмерный тензор — это просто «стопка» разностей по всем парам «точка–центроид»: первая ось выбирает точку b, вторая — центроид k, а третья хранит координаты по признакам $d = 1, \ldots, D$.

Чтобы получить `diff` *векторно*, без циклов, мы временно добавляем оси длины 1 так, чтобы формы вычитались по правилу broadcasting. В коде это записывается как

$$\texttt{X\_batch[:, None, :]} \in \mathbb{R}^{B\times 1\times D}, \qquad \texttt{centroids[None, :, :]} \in \mathbb{R}^{1\times K\times D}.$$

Далее поэлементное вычитание `X_batch[:, None, :]` − `centroids[None, :, :]` автоматически *транслирует* (broadcast) размерности: оси длины 1 «растягиваются» до нужных размеров, так что результат имеет форму (B, K, D) и эквивалентен «каждая из B строк X вычтена из всех K строк C». При этом PyTorch не копирует данные $B \times K$ раз — трансляция логическая, вычисления идут по правилу совместимости размерностей.

Трёхмерность здесь — промежуточный технический шаг, позволяющий за один векторизованный вызов получить все (b, k)-разности. Сразу после этого мы сворачиваем координатную ось D, чтобы получить матрицу квадратов расстояний:

$$\texttt{dist2} \;=\; (\texttt{diff}\cdot\texttt{diff}).\texttt{sum(dim=2)} \;\in\; \mathbb{R}^{B\times K}, \qquad \texttt{dist2}[b, k] \;=\; \sum_{d=1}^{D}(X_{b,d}-C_{k,d})^2 \;=\; \|x_b-c_k\|_2^2.$$

Интуитивно о вычитании с broadcasting и формах тензоров.
В батче $X_{\text{batch}} \in \mathbb{R}^{B\times D}$ каждая строка — это один объект с D признаками. В матрице центроидов $C \in \mathbb{R}^{K\times D}$ каждая строка — центр одного из K кластеров. Чтобы одновременно посчитать разности «объект минус центроид» по всем парам (b, k), мы добавляем «фиктивные» оси: берём $X_{\text{batch}}[:, \texttt{None}, :] \in \mathbb{R}^{B\times 1\times D}$ и $C[\texttt{None}, :, :] \in \mathbb{R}^{1\times K\times D}$. Правило broadcasting говорит: оси, где стоит 1, «растягиваются» до нужного размера при поэлементной операции, поэтому вычитание

$$\texttt{diff} \;=\; X_{\text{batch}}[:, \texttt{None}, :] \;-\; C[\texttt{None}, :, :]$$

даёт тензор формы $\mathbb{R}^{B\times K\times D}$, где элемент $\texttt{diff}[b, k, d] = X_{\text{batch}}[b, d] - C[k, d]$. Интуитивно это то же самое, как если бы мы концептуально «расплющили» пары в матрицу формы $(B{\cdot}K,\ D)$: по строкам идут все пары «объект–центроид», по столбцам — признаки. Мы так явно не делаем, а сразу сворачиваем признаковую ось D: считаем квадраты и суммируем по d, получая матрицу расстояний

$$\texttt{dist2} \;=\; (\texttt{diff}\cdot\texttt{diff}).\texttt{sum}(\texttt{dim} = 2) \;\in\; \mathbb{R}^{B\times K},$$

где в каждой строке b стоят расстояния текущего объекта до всех K центроидов, а в каждом столбце k — расстояния всех B объектов до данного центроида. На этом шаге мы имеем «карту расстояний»: на пересечении (b, k) — $\|x_b - c_k\|_2^2$. Далее применяем `softmax` *по оси кластеров* (в коде $\texttt{dim} = 1$) к отрицательным расстояниям с температурой τ:

$$q \;=\; \texttt{softmax}\big(-\,\texttt{dist2}/\tau,\ \texttt{dim} = 1\big) \in \mathbb{R}^{B\times K},$$

и получаем вероятности принадлежности каждого объекта к каждому клакстеру: в каждой строке b значения $q_{b,k} \in (0,1)$ складываются в 1. Таким образом, трёхмерный тензор $(B \times K \times D)$ — лишь удобный промежуточный формат, который позволяет векторно (без циклов) посчитать все попарные разности по признакам и затем свести их к нужной нам двумерной матрице расстояний $(B \times K)$.

Продолжаем обсуждение про softmax.
Далее вычисляются мягкие присваивания q — вероятности принадлежности точки x_b каждому кластеру k. Они получаются как softmax от *отрицательных* расстояний с температурой $\tau > 0$:

$$q_{b,k} \;=\; \frac{\exp\bigl(-d_{b,k}/\tau\bigr)}{\sum_{j=1}^{K}\exp\bigl(-d_{b,j}/\tau\bigr)}\,, \qquad q_b \in \Delta^{K-1},\ \sum_k q_{b,k} = 1.$$

Отрицательный знак перед $d_{b,k}$ принципиален: чем *ближе* точка к центру, тем *выше* вероятность. Параметр τ управляет «мягкостью»: при $\tau \to 0$ распределение становится почти onehot (классический жёсткий k-means. То есть расстояние между точками при деление на маленькое τ становится очень большим, из-за чего получаются большие различия в вероятностях), при больших τ — более равномерным (точка частично принадлежит нескольким кластерам. То есть вероятности получаются очень близкими друг к другу по значениям). В коде это строка `q = torch.softmax(-dist2 / tau, dim=1)`.

Лосс складывается из двух слагаемых. Первое — ожидаемое (по q) квадратичное расстояние до центров:

$$\text{recon} \;=\; \frac{1}{B}\sum_{b=1}^{B}\sum_{k=1}^{K} q_{b,k}\, d_{b,k}$$

и оно «наказывает» за далёкое расположение точек от выбранных центроидов. Второе — энтропия распределений q_b (со знаком минус в целевой функции из-за минуса, который дает логарифм):

$$H(q) \;=\; -\,\frac{1}{B}\sum_{b=1}^{B}\sum_{k=1}^{K} q_{b,k}\,\log(q_{b,k}+\varepsilon), \qquad \varepsilon = 10^{-12},$$

где маленькое ε обеспечивает численную устойчивость (избегает $\log 0$). Энтропия измеряет неопределённость: равномерные распределения имеют высокое H, почти onehot — низкое. Логарифм здесь показывает наше «удивление» из-за того, что точка с маленькой вероятностью попала имеено в этот кластер. То есть, если энтропия большая, то это показывает нам, что в кластере есть точки, которые скорее всего ему не принадлежат.

Итоговая цель:

$$L(C; X_{\text{batch}}) \;=\; \underbrace{\text{recon}}_{\text{среднее расстояние}} \;-\tau\; \underbrace{H(q)}_{\text{мягкость присваиваний}}\,.$$

Знак минус перед $H(q)$ выбран так, чтобы *поощрять* более высокую энтропию в начале обучения (мягкие присваивания предотвращают преждевременное «залипание» в плохую жёсткую сегментацию. То есть, если взять энтропию со знаком плюс, то это приведет к выбору точек с максимальной вероятностью попасть в кластер, из-за чего модель на ранних этапах может зафиксировать точки в плохо обученных кластерах) и постепенно,

по мере оптимизации центроидов, позволять распределениям становиться определённее. Параметр τ здесь играет двойную роль: он и «температура» в softmax, и вес энтропийного слагаемого, согласующий масштабы двух разных величин (по сути указывает, насколько сильно мы учитываем энтропию по отношению к расстоянию).

К вопросу о том, как меняется энтропия:.
Если вероятность отнесения ко всем кластерам примерно одинаковя, то значение энтропии удаляется от нуля и говорит нам, что точка весит где-то между кластерами. Если же вероятность отнесения к одному кластеру $\to 1$, а к другому $\to 0$, то первый член суммы зануляется из-за логарифма, а другой из-за вероятности, и это уже говорит нам об уверенности отнесения объекту к кластеру + уменьшает энтропию.

Заметим, что softmax — многоклассовый аналог сигмоиды: он преобразует вектор действительных чисел в вероятности, суммирующиеся к 1. Батч B — это просто подмножество объектов, обрабатываемых одновременно (в коде X_{batch} имеет форму (B, D)). Градиентная оптимизация меняет только центроиды C (они объявлены как обучаемый параметр `nn.Parameter`), стремясь минимизировать L; на практике это даёт гладкую, дифференцируемую версию k-means, к которой классический алгоритм приближается при $\tau \to 0$.

2.2 torch.optim.LBFGS

`LBFGS` — квази-ньютонинский метод для гладких целей. И в PyTorch мы реализуем его следующим образом:

```
C = nn.Parameter(init_centroids.clone().to(device), requires_grad=True)
opt = torch.optim.LBFGS([C], lr=1.0, max_iter=10, history_size=10,
                        line_search_fn="strong_wolfe")

def closure():
    opt.zero_grad(set_to_none=True)
    loss = soft_kmeans_loss(C, X_t, tau=0.5)  # full-batch
    loss.backward()
    return loss

for _ in range(max(1, 300 // 10)):
    loss = opt.step(closure)
```

Что такое `nn.Parameter` и почему `.clone().to(device)`. `nn` — это модуль `torch.nn` с «строительными блоками» нейросетей. `nn.Parameter` — подкласс `Tensor`, помечающий тензор как *обучаемый параметр*. Если такой параметр находится внутри `nn.Module`, он автоматически регистрируется и попадает в `model.parameters()` (это можно передать в `nn.Parameter` для оптимизации). В примере мы явно создаём параметр центроидов:

`C = nn.Parameter(init_centroids.clone().to(device), requires_grad=True).`

Здесь `.clone()` гарантирует, что `C` имеет собственное хранилище и является *листовым* тензором (без `grad_fn`), что важно для корректного подсчёта градиентов и работы оптимизатора. Вызов `.to(device)` переносит данные на нужное устройство (CPU/GPU) и приводит к нужному типу, чтобы устройство `C` совпадало с устройством данных `X_t`.

Почему именно `.clone().to(device)` и что значит «листовой» тензор.

- `.clone()`. Создаёт новый тензор с *собственным* хранилищем памяти, а не ссылку/«вид» на исходные данные. Это важно, чтобы параметр не зависел от чужих буферов и корректно участвовал в автодиффе как самостоятельный объект. В нашем кейсе исходные `init_centroids` созданы без градиентов, поэтому `clone()` даёт тензор без `grad_fn` (см. ниже), и после обёртки в `nn.Parameter(..., requires_grad=True)` он становится обучаемым «листом».
- `.to(device)`. Переносит тензор на нужное устройство (CPU/GPU) и, при необходимости, меняет тип данных: `.to(device)`, `.to(dtype=torch.float32)`, или сразу `.to(device, dtype=torch.float32)`. Это гарантирует совпадение устройства/типа у параметра C и данных X_t, иначе операции (вычитание, матричные умножения) упадут с ошибкой «tensors on different devices/dtypes».
- **«Листовой» (leaf) тензор.** Тензор называется *листовым*, если он *не* получен как результат дифференцируемой операции (у него `grad_fn=None`) и создан пользователем явно (или через `detach()`). Только у листовых тензоров при `backward()` накапливаются градиенты в поле `.grad`; именно такие тензоры оптимизатор умеет обновлять. Параметры моделей (`nn.Parameter`) по смыслу должны быть листовыми.
- `grad_fn`. Это ссылка на «функцию-источник» в графе автодифференцирования, породившую тензор. Если `grad_fn` не `None`, тензор — результат операции (например, `CloneBackward`, `AddBackward` и т.п.) и *не* является листом; градиент для него не сохраняется в `.grad` (он транзитно считается для его «родителей»). У листовых тензоров `grad_fn=None`.

`nn.Parameter`: какие аргументы вообще есть.

- **Подпись:** `nn.Parameter(data, requires_grad=True)`. У `Parameter` ровно *два* аргумента: исходный тензор `data` и флаг `requires_grad`. Других позиционных/именованных параметров у конструктора нет.
- `data`. Любой `torch.Tensor`. Если нужно задать устройство/тип, делайте это *до* обёртки: `torch.tensor(..., device=..., dtype=...)` или потом `data.to(...)`.
- `requires_grad`. Включает/выключает обучение данного параметра. Если `False`, градиенты не будут считаться, оптимизатор его не изменит — удобно для «заморозки» части модели.
- **Как задать прочее.** Параметры вроде `device`, `dtype`, форма, инициализация — задаются через сам `Tensor` (фабрики `torch.zeros/ones/empty`, `torch.randn`, `.uniform_()`, `.normal_()` и т.п.) *до* обёртки в `nn.Parameter`. Для групп параметров есть контейнеры `nn.ParameterList` и `nn.ParameterDict`.

`requires_grad`: зачем и как работает. Если `requires_grad=True`, то во время `loss.backward()` PyTorch вычислит $\partial L/\partial C$ и запишет в `C.grad`. Если `False`, градиент не считается (параметр «заморожен»). Это основной переключатель «обучаемости» для любого тензора/-параметра.

Как передавать параметры оптимизатору в общем случае.

- **Один параметр (как здесь):** `opt = torch.optim.SGD([C], lr=...)`.

- **Вся модель:** `opt = torch.optim.Adam(model.parameters(), lr=...)` — попадут все `nn.Parameter` внутри `model`.

Какие оптимизаторы есть в `torch.optim` (краткая ориентировка).

- **SGD** (с `momentum`, опц. `nesterov`) — простой и масштабируемый; ключевые параметры: `lr`, `momentum`, `weight_decay`.
- **Adam**/**AdamW** — адаптивные методы; `AdamW` с «декуплированным» `weight_decay`; ключевые параметры: `lr`, `betas`, `eps`, `weight_decay`.
- **RMSprop** — хорошо «гасит» шум за счёт скользящего среднего квадратов градиентов; ключевые: `lr`, `alpha`, `eps`.
- **Adagrad**/**Adadelta** — накапливают масштаб для каждого параметра; работают «из коробки», но шаг со временем затухает.
- **LBFGS** — квази-Ньютоновский метод для гладких задач; требует *closure*.
- Также есть **Adamax**, **ASGD**, **Rprop**, **NAdam**, **RAdam** — более узкие случаи/вариации.

`torch.optim.LBFGS`: ключевые параметры.

- `lr` — базовый коэффициент шага внешней итерации (грубая «скорость» метода).
- `max_iter` — *сколько внутренних итераций* сделать за один `step()`. В примере стоит `10`, а значит `step()` может несколько раз вызвать `closure()`. То есть внутри `step(closure)` функция `closure` может быть вызвана для оптимизации `loss` *не больше 10 раз*. (см. ниже)
- `max_eval` — максимум пересчётов целевой функции; по умолчанию выбирается из `max_iter`.
- `tolerance_grad`, `tolerance_change` — критерии остановки по норме градиента и по изменению функции (численные пороги).
- `history_size` — объём памяти L-BFGS (сколько последних направлений/«пар» (s, y) хранить для приближения Гессиана); типичные значения 10–100. Больше — точнее аппроксимация кривизны, но дороже по памяти.
- `line_search_fn` — стратегия линейного поиска. Допустимые значения: `None` (без явного поиска; используется внутренняя эвристика) или `"strong_wolfe"` (поиск длины шага, удовлетворяющей *сильным условиям Вольфа*: достаточное убывание/условие Армихо и условие кривизны; обычно стабильнее на негладких/шумных ландшафтах, но дороже — понадобится больше вызовов `closure()`).

Что делает `opt.step(closure)` у L-BFGS. Для L-BFGS `closure` обязателен: оптимизатор должен *многократно* пересчитывать значение лосса и его градиент, чтобы построить квази-Гессиан и провести (опционально) линейный поиск. Один `step()` — это «внешняя» итерация, внутри которой выполняется до `max_iter` «внутренних» итераций; на каждой из них L-BFGS вызывает `closure()`, читает `loss` и `.grad` у параметров, обновляет аппроксимацию Гессиана (через историю направлений) и выбирает шаг. Поэтому в примере внешний цикл ограничивает число `step()` так, чтобы общее «бюджетное» число внутренних шагов было около 300: `max(1, 300 // 10)`.

Что происходит внутри `closure()`.

- `opt.zero_grad(set_to_none=True)` — обнуляет (точнее, устанавливает в `None`) накопленные градиенты у всех параметров оптимизатора. В PyTorch градиенты *накапливаются* от вызова к вызову `backward()`; поэтому перед каждым новым пересчётом их нужно сбрасывать. Режим `set_to_none=True` экономит память и чуть быстрее, чем явная запись нулей; для оптимизаторов это эквивалент нулю.
- `loss = soft_kmeans_loss(C, X_t, tau=0.5)` — вычисляется текущее значение целевой функции на всех данных (full-batch).
- `loss.backward()` — автоматическое дифференцирование (autograd) вычисляет градиенты $\partial L/\partial C$ и складывает их в `C.grad`. По умолчанию граф не сохраняется (`retain_graph=False`); функция возвращает *скалярный* `loss`, который L-BFGS использует для критериев и линейного поиска.
- **Возврат `loss`.** `closure` обязан вернуть скалярную потерю; `L-BFGS` может вызывать `closure` многократно внутри одного `step()` до достижения условий останова (включая Strong Wolfe, если включён линейный поиск).

Почему L-BFGS требует `closure`, а Adam/SGD — нет. SGD/Adam делают один градиентный шаг на основе уже посчитанного градиента и не требуют переоценки лосса *внутри* `step()`. L-BFGS же строит аппроксимацию второй производной и подбирает длину шага; для этого ему нужно многократно «спросить» у модели актуальные `loss` и `grad` при разных кандидатов шага — именно это и делает `closure`.

Итог по строчке с параметром: `nn.Parameter(init_centroids.clone().to(device), requires_grad=True).` Это создаёт *листовой* обучаемый тензор центроидов $C \in \mathbb{R}^{K\times D}$ на нужном устройстве, независимый от исходного массива и готовый к оптимизации. Флаг `requires_grad=True` включает подсчёт градиентов для C. Передача `[C]` в оптимизатор делает этот параметр «управляемым» (его `.grad` будет обновляться и он будет изменяться вызовами `step()`).

2.3 torch.optim.Adam

`Adam` — адаптивный метод первого порядка, который поддерживает для каждого параметра свою «масштабированную» скорость за счёт экспоненциальных средних градиентов и их квадратов. Он устойчив к разномасштабным признакам (хотя стандартизация всё равно помогает). Управляющие параметры: `lr` (в ноутбуке 0.1 для `Iris`), `betas` (по умолчанию `(0.9, 0.999)`) и `eps` (по умолчанию `1e-8`). В примере `Adam` работает в *full-batch* режиме: каждый шаг видит все точки и обновляет центроиды один раз.

```
# Full-batch Adam
opt = torch.optim.Adam([C], lr=adam_lr)
for _ in range(steps):
    opt.zero_grad(set_to_none=True)
    loss = soft_kmeans_loss(C, X_t, tau=tau)  # full-batch
    loss.backward()
    opt.step()
    losses.append(float(loss.item()))
```

Что происходит:

- `opt = torch.optim.Adam([C], lr=adam_lr)` — создаём оптимизатор Adam, который будет обновлять параметр `C`. Квадратные скобки означают, что в оптимизатор передаётся список параметров/тензоров; шаг обучения задаётся `adam_lr`.
- `for _ in range(steps)` — внешний цикл оптимизации; столько раз параметр `C` будет обновлён.
- `opt.zero_grad(set_to_none=True)` — обнуляем накопленные градиенты у параметров, которыми управляет оптимизатор; флаг `set_to_none=True` делает градиенты `None` (экономия памяти и скорость).
- `loss = soft_kmeans_loss(C, X_t, tau=tau)` — считаем скалярный лосс по всей выборке (*full-batch*); см. разбор функции `soft_kmeans_loss`.
- `loss.backward()` — автодифференцирование: вычисляет ∂`loss`$/\partial$`C` и записывает его в `C.grad`.
- `opt.step()` — шаг Adam: обновляет `C` с учётом текущего градиента и внутренних моментов.
- `losses.append(float(loss.item()))` — сохраняем значение лосса как число Python для логирования/графика.

`loss.item()`: зачем и как работает.

- **Что это.** `item()` — метод PyTorch-тензора *скалярной* формы (0-D), который возвращает его численное значение как число Python (`float` или `int`). Пример: `float_loss = loss.item()`.
- **Когда уместно.** Для логирования/печати лосса после `backward()` и `step()`, чтобы сохранить значение в список/файл, построить график и т.п. Возврат — уже без вычислительного графа.
- **Важно:** Метод называется `item()` (без `s`). Варианта `items()` у тензоров нет — это опечатка (у словарей Python есть `dict.items()`, у тензоров — нет).
- **Требование к форме.** `item()` работает *только* для скаляров. Если тензор не скаляр (например, форма `(B,)` или `(B,K)`), будет ошибка. В таком случае используйте `tensor.tolist()` (список Python), `tensor.detach().cpu().numpy()` (NumPy) или предварительно усредните, чтобы получить скаляр.
- **Градиенты.** `item()` *отрывает* значение от графа (возвращает чистое число Python), поэтому по нему нельзя делать `backward()`. Это нормально для логирования, но не для дальнейших вычислений с автодифференцированием.
- **Производительность.** На GPU `item()` синхронизирует устройство с CPU (блокирует поток), поэтому не вызывайте его слишком часто внутри горячих циклов. Лучше суммировать/усреднять лосс тензорно и вызывать `item()` реже (например, раз в эпоху/итерацию внешнего цикла).

2.4 Финальный расчёт меток и центроидов (инференс без градиентов).

```
# Get hard labels on full data
with torch.no_grad():
    diff = X_t[:, None, :] - C[None, :, :]
    dist2 = (diff * diff).sum(dim=2)            # (N, K)
    q = torch.softmax(-dist2 / tau, dim=1)      # (N, K)
    labels = torch.argmax(q, dim=1).detach().cpu().numpy()
    C_cpu = C.detach().cpu().numpy()
```

`with torch.no_grad():` что даёт этот контекст.

- В режиме инференса нам не нужны градиенты и граф вычислений, поэтому контекст `torch.no_grad()` отключает отслеживание операций автодифференциатором, что уменьшает расход памяти и ускоряет вычисления, так как промежуточные тензоры не сохраняются для последующего `backward()`.

Зачем снова пересчитываются расстояния и softmax.

- После обучения оптимизатором у нас есть обновлённые центроиды `C`, поэтому нужно получить итог кластеризации на всём датасете: для каждой точки пересчитать расстояния до всех центроидов, превратить их в вероятности принадлежности с помощью `softmax` и на основе этих вероятностей выбрать итоговый кластер; это тот же самый расчёт, что внутри `soft_kmeans_loss`, только теперь на всей выборке сразу.

Формирование жёстких меток `labels`.

- Матрица `q` имеет размер $N \times K$, где каждая строка содержит вероятности принадлежности одной точки ко всем K кластерам, а `torch.argmax(q, dim=1)` берёт индекс максимальной вероятности в каждой строке и возвращает вектор длины N со значениями от 0 до $K-1$, далее `detach()` отрывает результат от графа, `cpu()` переносит данные на CPU, а `numpy()` преобразует тензор в массив NumPy для последующей визуализации или сохранения.

Сохранение центроидов в удобном формате `C_cpu`.

- Выражение `C_cpu = C.detach().cpu().numpy()` делает копию текущих центроидов без связи с графом, переносит их на CPU и преобразует в массив NumPy, чтобы можно было легко рисовать центры на графиках, логировать координаты или сохранять результаты на диск.

Итог блока.

- Финальная часть кода подводит результаты обучения: для каждой точки вычисляются вероятности принадлежности и выбирается жёсткая метка кластера, а также

готовятся к использованию на CPU финальные координаты центроидов в виде массивов NumPy; контекст `no_grad` делает этот этап быстрым и экономным по памяти, так как градиенты больше не нужны.

2.5 Метрика силуэта (`silhouette_score`).

```
# Когда мы внутри функции обращаемся к имени переменной, интерпретатор сначала ищет её во внутренней (локальной) области функции. Если не находит -- идёт в область видимости выше (глобальная область модуля).
sil = silhouette_score(X, labels) if len(np.unique(labels)) > 1 else np.nan
```

Силуэт измеряет, насколько хорошо объекты отделены между кластерами по сравнению с тем, как тесно они «свои» внутри кластера. Для каждого объекта i вычисляются две величины:

$$a(i) = \frac{1}{|C(i)| - 1} \sum_{j \in C(i),\, j \neq i} d(i, j) \quad \text{(средняя дистанция до } \textit{своего} \text{ кластера)},$$

$$b(i) = \min_{C \neq C(i)} \frac{1}{|C|} \sum_{j \in C} d(i, j) \quad \text{(минимальная средняя дистанция до } \textit{чужого} \text{ кластера)}.$$

Точка-силуэт:

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

Итоговая метрика по датасету — это среднее $S = \dfrac{1}{N} \sum_i s(i)$.

Интерпретация значений.

- $s(i) \approx 1$: объект хорошо вписан в свой кластер и далеко от других (кластеризация для него «чистая»).
- $s(i) \approx 0$: объект лежит на границе кластеров (равноудалён от своего и ближайшего чужого).
- $s(i) < 0$: объект ближе к чужому кластеру, чем к своему (часто признак ошибочного присвоения).

На практике средний силуэт S часто считают «хорошим» при $S \gtrsim 0.5$, «удовлетворительным» при $0.2 - 0.5$ и «слабым» при $S < 0.2$ (эвристика, не правило).

Когда метрика определена и зачем проверка в коде.

- Нужны *как минимум два* различных кластера в `labels`. Если кластер всего один, $b(i)$ не определено — потому в коде стоит проверка `len(np.unique(labels)) > 1`; иначе возвращаем `np.nan`.
- Кластеры из одного объекта допустимы: для такой точки $a(i) = 0$, силаэт всё равно считается (по формуле выше).

Как вызваны функции в `scikit-learn`.

- `silhouette_score(X, labels, metric='euclidean', sample_size=None, random_state=None` — средний силуэт по всем объектам (или по подвыборке, если задан `sample_size`).
- `silhouette_samples(X, labels, ...)` — покомпонентные $s(i)$, удобно для диагностики и графиков.
- `metric` по умолчанию `"euclidean"`; можно `"manhattan"`, `"cosine"` и др., либо `"precomputed"` для заранее посчитанной матрицы дистанций.

Важные практические замечания.

- Масштабирование признаков критично: при разных шкалах одна ось может доминировать в расстояниях, и силуэт исказится.
- Стоимость по времени/памяти $\mathcal{O}(N^2)$ из-за попарных дистанций; для больших N используйте `sample_size`.
- Силуэт лучше отражает качество *выпуклых, примерно равноплотных* кластеров; для вытянутых/не-выпуклых структур (типа «полумесяцев») он может вводить в заблуждение.
- Если есть «шум» с меткой -1 (например, после DBSCAN), `silhouette_score` трактует -1 как обычный кластер; обычно шум стоит отфильтровать перед расчётом силуэта.

Мини-пример использования.

```python
from sklearn.metrics import silhouette_score, silhouette_samples

# X: (N, d), labels: (N,), k >= 2 уникальных значений в labels
S = silhouette_score(X, labels, metric="euclidean")
s_per_point = silhouette_samples(X, labels)

print("average silhouette:", S)
print("per-sample silhouette shape:", s_per_point.shape)
```

Функция `silhouette_samples`: покомпонентный силуэт.

- `silhouette_samples(X, labels, metric='euclidean', metric_params=None)` возвращает вектор $s \in [-1, 1]^N$, где каждая компонента $s(i)$ — это силуэт *конкретного* объекта i. Формула такая же, как у среднего силуэта: для точки i вычисляются $a(i)$ — среднее расстояние до всех точек *своего* кластера, и $b(i)$ — минимальная средняя дистанция до *ближайшего чужого* кластера; затем $s(i) = \dfrac{b(i) - a(i)}{\max\{a(i),\, b(i)\}}$. Положительные значения (ближе к 1) означают «правильное» присвоение, около нуля — пограничные случаи, отрицательные — потенциальные ошибки кластеризации.

Когда использовать.

- Эта функция нужна для диагностики: построить «силуэтные» диаграммы по каждому кластеру, выявить точки с $s(i) < 0$, сравнить распределения $s(i)$ между кластерами, оценить неоднородность кластеров. В отличие от `silhouette_score`, она не усредняет, а возвращает значения по всем объектам.

Параметры и возвращаемое значение.

- `X` — матрица признаков формы (N, d) или попарная матрица дистанций (N, N), если задано `metric="precomputed"`.
- `labels` — вектор меток кластеров длины N (целые номера кластеров); должно быть ≥ 2 различных значений.
- `metric` — метрика расстояний: по умолчанию `"euclidean"`; допустимы `"manhattan"`, `"cosine"` и др., а также `"precomputed"`.
- `metric_params` — опциональный словарь доп. параметров выбранной метрики (если требуется).
- **Возврат** — `ndarray` формы $(N,)$ с силуэтами каждого объекта.

Как читать и использовать значения.

- $s(i) \approx 1$ — объект хорошо «сидит» в своём кластере и далеко от остальных; $s(i) \approx 0$ — на границе; $s(i) < 0$ — вероятная ошибочная принадлежность.
- Для быстрой диагностики смотрят $\min s(i)$, долю отрицательных $s(i)$, а также среднее $s(i)$ по каждому кластеру (может отличаться от глобального среднего).
- Вычислительная стоимость $\mathcal{O}(N^2)$ по памяти/времени из-за попарных расстояний; для очень больших N стройте графики/статистики по подвыборке.

2.6 Что возвращает функция (словарь результатов).

Функция возвращает обычный `dict` Python с агрегированными итогами одного запуска оптимизатора; ключи и значения следующие:

- `"name"` — строка: имя использованного оптимизатора (`"LBFGS"` или `"Adam"`).
- `"centroids"` — `numpy.ndarray` формы (K, D): финальные координаты центроидов в пространстве признаков. Каждая строка — один кластер, каждый столбец — признак.
- `"labels"` — `numpy.ndarray` формы $(N,)$ целых чисел: «жёсткие» метки кластеров для всех объектов. Значения в диапазоне $[0, K-1]$ и соответствуют строкам `X` в исходном порядке.
- `"losses"` — `numpy.ndarray` формы $(T,)$ с траекторией лосса по шагам обучения: для `Adam` — по внешним шагам (эпохам), для `LBFGS` — по внешним вызовам `step()` (внутренние итерации считаются внутри оптимизатора и сюда не попадают). Полезно для кривых сходимости.
- `"silhouette"` — число с плавающей точкой: итоговый `silhouette_score` на всём датасете при финальных метках; если кластеров оказалось меньше двух, возвращается `np.nan`.

2.7 Короткие практические выводы

Если цель гладкая и размер данных умеренный, `LBFGS` в *full-batch* часто даёт быстрые и стабильные шаги. `Adam` хорошо стартует и менее чувствителен к выбору шага, но при долгой тренировке может потребовать уменьшения `lr`. `SGD` масштабируется и даёт гибкость через `batch_size`, однако требует аккуратной настройки `lr` и перемешивания данных. Для линейной классификации по табличным данным `LogisticRegression` с уменьшением `C` — крепкий базовый вариант; `SGDClassifier` полезен на больших наборах; `AdamW` в `PyTorch` удобен, когда хочется явно контролировать `weight_decay` (регуляризацию) и интегрировать модель в тензорный пайплайн.

2.8 Мини-пример: настройка регуляризации и оптимизация модели (PyTorch, Adam).

```python
import numpy as np
import torch
from torch import nn

device = "cuda" if torch.cuda.is_available() else "cpu"
Xt = torch.tensor(X, dtype=torch.float32, device=device)
yt = torch.tensor(y, dtype=torch.float32, device=device).view(-1, 1)

model = nn.Sequential(nn.Linear(2, 1)).to(device)
criterion = nn.BCEWithLogitsLoss()

# Базовый Adam (без L2). Для L2 используйте weight_decay.
# opt = torch.optim.Adam(model.parameters(), lr=1e-1)
# Пример с L2-пенальти (coupled) прямо в Adam:
opt = torch.optim.Adam(model.parameters(), lr=1e-2, weight_decay=1e-2)


losses = []
for it in range(300):
    opt.zero_grad()             # 1) обнуляем прошлые градиенты
    logits = model(Xt)          # 2) прямой проход -> логиты z \in R^{N\times 1}
    loss = criterion(logits, yt)# 3) BCEWithLogitsLoss: считает BCE по логитам
    loss.backward()             # 4) автоград: dL/dw
    opt.step()                  # 5) шаг оптимизатора (обновление весов)
    losses.append(loss.item())  # 6) сохраняем скалярное значение лосса

print("PyTorch Adam final loss (BCE):", losses[-1])

# Инференс: перевод логитов в вероятности сигмоидой
def torch_predict_proba(Z):
    Zt = torch.tensor(Z, dtype=torch.float32, device=device)
    with torch.no_grad():                           # отключаем граф градиентов
        logits = model(Zt).cpu().numpy().ravel()  # logits -> numpy, shape: (N,)
    return 1.0 / (1.0 + np.exp(-logits))          # sigma(z)
```

Что делает код по шагам.
Сначала подключаются `torch` и `nn`, выбирается устройство `device` (GPU при наличии), и исходные массивы `X`, `y` приводятся к тензорам `float32` на том же устройстве; целевой вектор `y` разворачивается до формы $(N, 1)$, как ожидает `BCEWithLogitsLoss`. Модель — это `nn.Sequential` из одного линейного слоя `nn.Linear(2,1)` (вход из двух признаков, выход — один логит). Функция потерь `BCEWithLogitsLoss` принимает *логиты* (а не вероятности) и стабильно внутри применяет сигмоиду. Оптимизатор `Adam` на каждом шаге: обнуляет градиенты `zero_grad()`, делает прямой проход (получаем логиты), считает лосс, выполняет обратное распространение `backward()` и обновляет параметры `step()`. Значение `loss.item()` добавляется в список `losses` для последующего анализа сходимости. Для инференса определена `torch_predict_proba`: внутри `no_grad` логиты переводятся в NumPy и через сигмоиду получаются вероятности $P(y = 1 \mid x)$.

2.8.1 Контейнер `nn.Sequential`: что это, что внутрь кладут, есть ли свои параметры

Идея. `nn.Sequential` — это *линейный* контейнер слоёв: он получает тензор на вход и прогоняет его последовательно через вложенные модули. Сам по себе `Sequential` *не имеет обучаемых параметров* — все параметры принадлежат вложенным слоям. Важно: каждый вложенный модуль должен принимать один вход и возвращать один выход, совместимый по форме со следующим модулем в цепочке.

Есть ли у `nn.Sequential` собственные параметры? Обучаемых гиперпараметров у `Sequential` нет: он просто хранит и вызывает дочерние модули. Тем не менее, как и любой `nn.Module`, он:

- наследует методы `.parameters()`, `.named_parameters()`, `.modules()`, `.children()` — для доступа к параметрам/подмодулям;
- умеет переключаться между режимами `train()`/`eval()` (влияет на Dropout/BatchNorm внутри);
- переносится на устройство/тип через `.to(device[, dtype])`, `.cuda()`, `.cpu()`, `.float()` и т.д.;
- поддерживает индексацию и слайсы: `seq[0]`, `seq[:2]` возвращают подцепочку;
- позволяет добавлять слои программно: `seq.add_module("name layer)` или создавать с `OrderedDict`.

Когда `Sequential` не подойдёт. Если в архитектуре нужны ветвления, суммирования/конкатенации путей, скип-соединения (residual), условные вычисления, несколько входов/выходов или модули со сложной сигнатурой `forward`, используйте собственный класс, унаследованный от `nn.Module`, и реализуйте логику в `forward`. `Sequential` оставьте для действительно «прямых» цепочек.

2.8.2 Что такое логиты (logits)

Определение (бинарный случай). В моделях классификации *логит* — это сырой выход модели до применения сигмоиды. Обозначим его через $z \in \mathbb{R}$. Связь с вероятностью положительного класса:

$$p \;=\; \sigma(z) \;=\; \frac{1}{1+e^{-z}}, \qquad z \;=\; \log\frac{p}{1-p} \;=\; \operatorname{logit}(p).$$

То есть логит — это *логарифм отношения шансов* (log-odds). Знак z определяет сторону порога (при $z = 0 \Rightarrow p = 0.5$), а модуль $|z|$ соответствует «уверенности» модели.

Определение (многоклассовый случай). При K классах модель выдаёт вектор логитов $z \in \mathbb{R}^K$ (по одному на класс). Вероятности получаются softmax-преобразованием:

$$p_k \;=\; \frac{e^{z_k}}{\sum_{j=1}^{K} e^{z_j}}, \qquad k = 1, \ldots, K.$$

Важно: добавление одной и той же константы ко всем компонентам z *не меняет* p (инвариантность softmax к сдвигу).

Интерпретация и свойства.

- **Неограниченность:** логиты лежат на всей прямой $\mathbb{R}$ (или в $\mathbb{R}^K$); вероятности всегда в $(0, 1)$ и суммируются к 1 (для softmax).
- **Порог и уверенность:** $z = 0 \Rightarrow p = 0.5$. Чем больше $|z|$, тем ближе p к 0 или 1 (большая уверенность).
- **Линейная форма:** в линейных моделях $z = w^\top x + b$ — скалярный (для бинарной) или набор линейных оценок (для многоклассовой).
- **Разности логитов:** при softmax именно разности $z_k - z_j$ определяют относительные шансы классов (логарифм отношения вероятностей).

Почему многие лоссы принимают *логиты*, а не вероятности. Функции `BCEWithLogitsLoss` (бинарная кросс-энтропия на логитах) и `CrossEntropyLoss` (многоклассовая кросс-энтропия) ожидают *не нормированные* логиты и *внутри* выполняют численно устойчивые операции (`log_sigmoid`, `logsumexp`). Это:

- уменьшает численные переполнения/потери точности при очень больших $|z|$;
- даёт стабильные градиенты, особенно в хвостах распределений;
- избавляет от необходимости вручную применять $\sigma(\cdot)$ или softmax$(\cdot)$ перед лоссом.

Практическое правило: *если используете `BCEWithLogitsLoss`/`CrossEntropyLoss`, не применяйте `sigmoid`/`softmax` к выходам модели до лосса.*

Связь с API (`scikit-learn`, `PyTorch`).

- В `sklearn` метод `decision_function(X)` возвращает «сырые оценки» z (логиты для логистической регрессии); `predict_proba(X)` — уже вероятности.
- В `PyTorch` модуль выдаёт логиты; вероятности можно получить `torch.sigmoid(z)` (бинарный) или `torch.softmax(z, dim=1)` (многоклассовый). Для обучения используйте `BCEWithLogitsLoss`/`CrossEntropyLoss`.

Мини-примеры.

```
# Бинарный случай: логиты -> вероятность
logits = model(X)                  # shape: (N, 1)
proba  = torch.sigmoid(logits)     # (0,1)

# Многоклассовый случай: логиты -> вероятности
logits = model(X)                  # shape: (N, K)
probs  = torch.softmax(logits, dim=1)  # строки суммируются к 1
```

2.8.3 `nn.Linear`: что это, как работает и зачем нужен

Суть. `nn.Linear` — это модуль PyTorch, реализующий *аффинное* (линейное со сдвигом) преобразование над последним измерением входного тензора:

$$y \;=\; x\,W^\top + b,$$

где $x \in \mathbb{R}^{*\times d_{\text{in}}}$ (звёздочкой обозначены любые ведущие размерности — батчи и пр.), $W \in \mathbb{R}^{d_{\text{out}}\times d_{\text{in}}}$, $b \in \mathbb{R}^{d_{\text{out}}}$, а на выходе получаем $y \in \mathbb{R}^{*\times d_{\text{out}}}$. Это самый базовый «полносвязный» слой: он умножает каждый вектор признаков на матрицу весов и добавляет смещение.

Формальные параметры конструктора. `nn.Linear(in_features, out_features, bias=True, device=None, dtype=None)`.

- `in_features` — размер входного признакового вектора d_{in}.
- `out_features` — размер выходного вектора d_{out}.
- `bias` — добавлять ли смещение b (по умолчанию `True`); если в модели сразу после `Linear` идёт нормализация (например, `BatchNorm`), смещение иногда отключают.
- `device`, `dtype` — на каком устройстве и с каким типом создать параметры (обычно задают позже через `.to(device)`).

Формы входов и выходов. Слой применяет преобразование *к последнему измерению* входа. Если на вход подан:

- 2D тензор формы (N, d_{in}) — получим (N, d_{out}) (классический батч).
- ND тензор $(n_1, \ldots, n_k, d_{\text{in}})$ — получим $(n_1, \ldots, n_k, d_{\text{out}})$; все ведущие измерения сохраняются, линейное преобразование применяется независимо к каждому вектору длины d_{in}.

Параметры и их число. Внутри `nn.Linear` обучаемые параметры — это `weight` формы $(d_{\text{out}}, d_{\text{in}})$ и, при `bias=True`, `bias` формы (d_{out}). Общее число параметров:

$$d_{\text{out}} \cdot d_{\text{in}} \; + \; \{\text{bias}\} \cdot d_{\text{out}}.$$

Частые сценарии использования.

- **Логистическая регрессия:** `nn.Linear(d,1)` даёт логит z; вероятность получают сигмоидой, а лосс — через `nn.BCEWithLogitsLoss()`.
- **Классификатор на K классов:** `nn.Linear(d,K)` выдаёт логиты (N, K); сверху — `nn.CrossEntropyLoss()` (внутри есть `log_softmax`).
- **Головы нейросетей:** после свёрточного экстрактора признаков (N, C, H, W) делают `Flatten()` и несколько `Linear` со слоями активации.

Практические мелочи и типовые ошибки.

- **Несоответствие форм:** последняя размерность входа должна равняться `in_features`.
- **Дублирование смещения:** если за `Linear` следует `BatchNorm1d`, часто ставят `bias=False` (смещение компенсирует BN).
- **Заморозка слоя:** чтобы не обучать слой, отключите градиенты: `for p in layer.parameters(): p.requires_grad=False`.
- **Устройство/тип:** переносите слой и данные на одно устройство и используйте согласованные `dtype` (`float32` — безопасный дефолт).

2.8.4 `nn.BCEWithLogitsLoss`: что это и как работает

Задача и идея. `BCEWithLogitsLoss` — это *бинарная кросс-энтропия на логитах*, стандартная функция потерь для бинарной классификации и *мульти-лейбл* задач (когда на один объект может приходиться несколько независимых меток). Она принимает *логиты* x (сырые выходы модели до сигмоиды) и целевые значения $y \in \{0, 1\}$ (или $[0, 1]$ при сглаживании/soft-лейблах) и возвращает скалярную ошибку.

Формула (один пример, один выход). Пусть $x \in \mathbb{R}$ — логит, $y \in \{0, 1\}$ — целевая метка. Наивная форма через сигмоиду $\sigma(x) = \frac{1}{1+e^{-x}}$:

$$\ell(x, y) \;=\; -\big(y\, \log \sigma(x) + (1 - y)\, \log(1 - \sigma(x))\big).$$

В реализации используется *численно устойчивая* эквивалентность (без явной сигмоиды):

$$\ell(x, y) \;=\; \max(x, 0) \;-\; x\, y \;+\; \log\big(1 + e^{-|x|}\big),$$

что предотвращает переполнения при больших $|x|$.

Бинарная кросс-энтропия: $-\big(y \log p + (1 - y) \log(1 - p)\big)$

Что считает и зачем. Эта функция потерь измеряет расхождение между истинной меткой $y \in \{0, 1\}$ и предсказанной моделью вероятностью $p \in (0, 1)$.

$$\ell(y, p) \;=\; -\Big(y \log p \;+\; (1 - y) \log(1 - p)\Big).$$

- Если $y = 1$, остаётся $-\log p$: чем ближе p к 1, тем меньше потеря; уверенная ошибка ($p \approx 0$) даёт огромный штраф.
- Если $y = 0$, остаётся $-\log(1 - p)$: чем ближе p к 0, тем лучше; уверенная ошибка ($p \approx 1$) снова штрафуется сильно.
- Лосс ℓ всегда ≥ 0, минимален при «правильной» уверенности модели.
- Грубо говоря: эта *loss-функция* указывает на различия между двумя распредлениями (инстинным и предсказанным). Истинное распределение умножается на логарифм предсказанного. Получается, что *loss-функция* штрафует модель за неуверенность в правильном ответе или за уверенность в неправильном.

Интуиция. Функция $-\log(\cdot)$ быстро растёт около нуля, поэтому *уверенные ошибки* наказываются значительно сильнее, чем неуверенные. Это стимулирует модель не только угадать класс, но и давать к нему высокую (обоснованную) вероятность.

Связь с логитами. На практике модель выдаёт *логит* $x \in \mathbb{R}$, а вероятность получается как $\sigma(x) = \frac{1}{1+e^{-x}}$. Тогда $\ell(y, p)$ можно писать как $\ell(y, \sigma(x))$.

Численно устойчивая эквивалентность (без явной сигмоиды). Чтобы избежать переполнений при больших $|x|$ и вычисления $\log(0)$, используют эквивалентную форму через `softplus`:

$$\ell(x, y) \;=\; \max(x, 0) \;-\; x\,y \;+\; \log\bigl(1 + e^{-|x|}\bigr).$$

Эта формула даёт те же значения, что и $-\bigl(y \log \sigma(x) + (1-y) \log(1-\sigma(x))\bigr)$, но считается стабильно: экспоненты не «взрываются», а логарифмы не получают нулевые аргументы.

Мини-пример. Пусть $y = 1$.

- Если $x = 4 \Rightarrow p \approx 0.982$, то $\ell \approx -\log 0.982 \approx 0.018$ — почти нет ошибки.
- Если $x = -4 \Rightarrow p \approx 0.018$, то $\ell \approx -\log 0.018 \approx 4.02$ — большая «цена» уверенной ошибки.

Ключевые параметры конструктора.

- `weight`: тензор весов. Масштабирует вклад отдельных элементов/примеров в среднее.
- `pos_weight`: тензор длины C (или скаляр для одного выхода), *усиливает* вклад *положительных* примеров класса(ов). Полезно при дисбалансе: `pos_weight` > 1 повышает цену ошибок на классе «1».
- `reduction`: `"mean"` (по умолчанию; усреднить по всем элементам), `"sum"` (просуммировать), `"none"` (вернуть лосс той же формы, что вход).

Когда использовать.

- **Бинарная классификация:** один выход $(N, 1)$ или $(N,)$; цель $(N, 1)/(N,)$.
- **Мульти-лейбл классификация:** выход (N, C), цель (N, C), каждая колонка — независимая «бинарка».
- Для *взаимоисключающих* K классов ($K > 2$) используйте `CrossEntropyLoss` (она сама применяет `log_softmax`).

Типичные ошибки и как их избежать.

- **Не применяйте `sigmoid` перед лоссом.** `BCEWithLogitsLoss` уже «знает» про сигмоиду внутри.
- **Типы и формы:** цели должны быть `float32`, а не целочисленные метки `long`. Формы входа и цели должны совпадать (или быть совместимыми по broadcasting).
- **Дисбаланс классов:** используйте `pos_weight` (весит *положительные* наблюдения), не путайте с `weight`, который масштабирует элементы в целом.

Мини-пример (бинарный случай, с `pos_weight`).

```python
import torch
from torch import nn

# Данные (N=5): логиты модели и бинарные цели
logits = torch.tensor([ 2.0, -1.0, 0.0, 3.0, -4.0])      # shape: (5,)
targets = torch.tensor([ 1.0,  0.0, 1.0, 1.0,  0.0])     # float!

# Усилим вклад положительных примеров (например, класс "1" редок)
criterion = nn.BCEWithLogitsLoss(pos_weight=torch.tensor(3.0))

loss = criterion(logits, targets)
print(float(loss))  # скалярная ошибка

# Вероятности для анализа/метрик (вне лосса)
proba = torch.sigmoid(logits)  # в (0,1)
```

Мини-пример (тренировочный шаг для линейной модели).

```python
model = nn.Sequential(nn.Linear(d, 1))  # выход: логиты
criterion = nn.BCEWithLogitsLoss()
opt = torch.optim.AdamW(model.parameters(), lr=1e-3, weight_decay=1e-2)

for _ in range(300):
    opt.zero_grad()
    logits = model(X)            # shape: (N, 1)
    loss = criterion(logits, y)  # y: shape (N, 1), float {0,1}
    loss.backward()
    opt.step()
```

Зачем «на логитах». Работа «на логитах» (вместо вероятностей) позволяет применять устойчивые преобразования (`logsigmoid`, `logsumexp`), что:

- предотвращает $\log(0)$ и переполнения при больших $|x|$;
- улучшает стабильность и качество градиентов;
- упрощает код (не нужно явно вызывать `sigmoid` перед лоссом).

Регуляризация: как настроить. Вариантов два. Во-первых, можно использовать параметр `weight_decay` в `torch.optim.Adam`; это добавляет L_2-штраф *сцеплённым* образом (coupled), т.е. через модификацию градиента. Во-вторых, предпочтительный современный способ — `torch.optim.AdamW`, где L_2 реализован *декуплированно* (weight decay отдельно от градиента), что даёт более корректное поведение при адаптивных шагах. Типичный рабочий диапазон: `weight_decay` $\in \{10^{-3}, 10^{-2}, 10^{-1}\}$, шаг `lr` начните с $10^{-3} \dots 10^{-2}$ и подберите по кривой `losses`. При слишком большом `weight_decay` появится недообучение (лосс «застынет» высоко), при слишком маленьком — риск переобучения.

Почему `BCEWithLogitsLoss`. Эта функция сразу принимает логиты $z = w^\top x + b$ и внутри вычисляет $\sigma(z)$, избегая численных переполнений при больших $|z|$. Если бы вы использовали `BCELoss`, пришлось бы самим применять `sigmoid()`, и тогда при больших логитах можно получить $\log(0)$ и NaN.

Частые проверки. Убедитесь, что формы согласованы: `Xt.shape=(N,2)`, `yt.shape=(N,1)`; типы — `float32`; все тензоры на одном устройстве (`cpu` или `cuda`). Если лосс не убывает или «взрывается», уменьшите `lr` (например, в 10 раз) и/или снизьте `weight_decay`; для стабильности входы лучше стандартизовать.

Функция `torch_predict_proba` (инференс вероятностей)

```
def torch_predict_proba(Z):
    Zt = torch.tensor(Z, dtype=torch.float32, device=device)
    with torch.no_grad():
        logits = model(Zt).cpu().numpy().ravel()
    return 1.0 / (1.0 + np.exp(-logits))
```

Коротко о том, что происходит.

- Вход `Z`: массив признаков формы `(M, d)` (обычно `numpy.ndarray`).
- `torch.tensor(..., dtype=torch.float32, device=device)` — создаём тензор нужного типа на CPU/GPU, как у модели.
- Блок `with torch.no_grad():` отключает автодифференцирование для быстрого и экономного по памяти инференса.
- `logits = model(Zt)` — модель возвращает логиты формы `(M, 1)`; `.cpu().numpy().ravel()` переносит на CPU и делает вектор `(M,)`.
- Возвращаем вероятность положительного класса через сигмоиду: $\sigma(x) = 1/(1 + e^{-x})$.

Получение меток классов по порогу

```
# получаем метки классов для каждого объекта
with torch.no_grad():
    logits = model(Xt)                      # (N, 1) -- логиты
    probs = torch.sigmoid(logits).squeeze(1)  # (N,) -- вероятности P(y=1|x)
    preds = (probs >= 0.5).long()           # (N,) -- метки {0,1} типа int64
```

Коротко о строках кода.

- `with torch.no_grad():` — режим инференса: не строим граф градиентов.
- `logits = model(Xt)` — прямой проход; на выходе сырые оценки (логиты) формы `(N,1)`.
- `torch.sigmoid(logits)` — переводим логиты в вероятности $\in (0, 1)$; `.squeeze(1)` убирает ось размера $1 \Rightarrow$ `(N,)`.
- `(probs >= 0.5)` — порог 0.5 даёт булевы метки; `.long()` превращает их в целочисленные `{0,1}` (`torch.int64`).

Примечание. Порог `0.5` — дефолт для симметричных задач. Его имеет смысл сдвигать (например, на `0.3`/`0.7`) при дисбалансе классов или если целевая метрика — `recall`/`precision`/`F1`.

Итог: что именно делает модель в коде (Logistic + Adam в PyTorch)

1) Архитектура и выход модели. Модель `nn.Sequential(nn.Linear(2,1))` реализует линейное отображение

$$z \;=\; w_1x_1 + w_2x_2 + b \;=\; w^\top x + b,$$

где z — *логит* (logit), то есть логарифм отношения шансов: $z = \log \frac{p}{1-p}$. На выходе слоя `nn.Linear` сразу получаем логиты (ещё не вероятности). Изначально смысл логита заключается в том, что он переводит вероятность $p \in (0,1)$ в вещественную прямую:

$$\operatorname{logit}(p) = \log\left(\frac{p}{1-p}\right) \in (-\infty, +\infty).$$

, что по смыслу и делает модель. На первых шагах полученные значения еще не совсем логиты, но по ходу оптимизации они ими становятся.

2) Инициализация параметров. При создании `nn.Linear(2,1)` веса $w \in \mathbb{R}^2$ и смещение $b \in \mathbb{R}$ инициализируются случайно (по умолчанию равномерно в $\mathcal{U}(-1/\sqrt{2},\, 1/\sqrt{2})$ в знаменателе стоит количество признаков, подаваемых в модель); это даёт ненулевую стартовую точку, от которой оптимизатор начинает подстройку. Затем модель переносится на устройство `device` (CPU/GPU).

3) Почему «логиты» и где берутся вероятности. В бинарной классификации удобно моделировать именно логиты z, а вероятность положительного класса получать через сигмоиду:

$$p(y=1 \mid x) \;=\; \sigma(z) \;=\; \frac{1}{1+e^{-z}}.$$

Сигмоида переводит любые $z \in \mathbb{R}$ в $p \in (0,1)$, что естественно интерпретировать как вероятность и удобно дифференцировать.

4) Функция потерь. Используется `BCEWithLogitsLoss()`, которая *внутри* применяет сигмоиду к логитам и считает бинарную кросс-энтропию (численно устойчивый вариант):

$$L \;=\; -\frac{1}{N}\sum_{i=1}^{N}\Big(y_i \log p_i + (1-y_i)\log(1-p_i)\Big), \quad p_i = \sigma(z_i).$$

Эта функция совместима с логитами на входе и корректно даёт градиенты для обучения.

5) Оптимизация (Adam). Оптимизатор `Adam` обновляет w, b по градиентам, используя адаптивные шаги (первые/вторые моменты градиента). Один шаг тренинга выглядит так:

1. `opt.zero_grad()` — обнулить старые градиенты.
2. `logits = model(Xt)` — прямой проход: получить логиты z.
3. `loss = criterion(logits, yt)` — посчитать BCE-лосс.

4. `loss.backward()` — автодифференцирование: $\nabla_w L, \nabla_b L$.
5. `opt.step()` — обновить параметры w, b.

Цикл повторяется 300 итераций; `loss.item()` сохраняется для мониторинга сходимости.

6) Как из логитов получить решения. После обучения модель выдаёт логиты z; для вероятностей применяем $\sigma(z)$. Для меток берём порог, обычно 0.5: если $p \geq 0.5 \Rightarrow y = 1$, иначе $y = 0$. При дисбалансе классов порог можно сдвигать.

7) Интуитивная картина. Модель учится подбирать w, b так, чтобы гиперплоскость $w^\top x + b = 0$ разделяла классы: на позитивной стороне сигмоида даёт p близко к 1, на негативной — к 0. Случайная инициализация задаёт стартовую прямую; `BCEWithLogitsLoss` и `Adam` итеративно вращают/сдвигают её, минимизируя ошибку.

3 Оптимизаторы sklearn: SGDClassifier и SGDRegressor

Идея. Семейство моделей `SGD*` обучает *линейные* модели (гиперплоскость $w^\top x + b$) стохастическим градиентным спуском по одному объекту или по минибатчам. Это даёт масштабируемость и позволяет использовать регуляризацию (L_2/L_1/elastic net) и разные схемы шага.

3.1 SGDClassifier: линейный классификатор

Назначение. Поддерживает бинарную и многоклассовую классификацию. Доступны SVM-подобные лоссы (`hinge`) и логистический `log_loss` с вероятностями.

Ключевые параметры (с допустимыми значениями).

- `loss` — функция потерь:
  - `"log_loss"` – то же, что и кросс=энтропия *(см. выше)* (логистическая; доступны `predict_proba` (пропускает сырые значения через сигмоиду)/`predict_log_proba`);
  - `"hinge"`, `"squared_hinge"` (линейный SVM, вероятностей нет);
  - `"modified_huber"` (робастная классификация; есть `predict_proba`);
  - `"perceptron"` (знак отступа).
- `penalty` — регуляризация: `"l2"`, `"l1"`, `"elasticnet"` или `None`.
- `alpha` — сила регуляризации (> 0); типично $10^{-6} \dots 10^{-3}$.
- `l1_ratio` — доля L_1 в `"elasticnet"` (интервал $0 \dots 1$).
- `learning_rate` — схема шага: `"optimal"`, `"constant"`, `"invscaling"`, `"adaptive"`.
- `eta0` — базовый шаг (для `"constant"`, `"invscaling"`, `"adaptive"`).
- `power_t` — показатель в `"invscaling"` ($\eta_t = \eta_0\, t^{-\texttt{power\_t}}$).
- `max_iter`, `tol` — число эпох и ранняя остановка.
- `shuffle` — перемешивать ли объекты между эпохами (`True` по умолчанию).
- `class_weight` — веса классов (`"balanced"` или словарь).
- `average` — усреднение весов (True/целое) для стабилизации на потоках.
- `random_state`, `fit_intercept`, `early_stopping`, `validation_fraction`, `n_iter_no_change`.

3.2 Допустимые значения параметров: loss и elasticnet

`loss` в `SGDClassifier`: `"hinge"`, `"squared_hinge"`, `"modified_huber"`

`"hinge"` **(линейный SVM).** Кусочно-линейная маржинальная потеря

$$\ell_{\text{hinge}}(y, f) = \max\bigl(0,\, 1 - y\, f\bigr).$$

Нулевая при правильной классификации с запасом ($y\, f \geq 1$), линейно растёт при нарушении отступа. Обучает «разделяющую гиперплоскость» с максимальным зазором. Вероятности не определены; используйте `decision_function`.

`"squared_hinge"` **(квадратичный hinge).** Квадратичный вариант маржинальной потери

$$\ell_{\text{s-hinge}}(y, f) = \big(\max(0,\, 1 - y\, f)\big)^2.$$

Сильнее штрафует крупные нарушения отступа, имеет непрерывный градиент на границе. Вероятности также не поддерживаются; интерпретируйте баллы через `decision_function`.

`"modified_huber"` **(сглажённая и робастная).** Робастная (испоьзуется для повышения устойчивости модели при шумных данных, данных с выбросами или ошибками) сглажённая альтернатива hinge: около границы ведёт себя как `squared_hinge`, а для «сильно неверных» примеров растёт линейно (ограничивает величину градиента, что повышает устойчивость к выбросам). В реализации `sklearn` для `loss="modified_huber"` доступны `predict_proba` (получаются из откалиброванных отступов), что удобно, когда нужны «вероятностные» оценки при SVM-подобной постановке.

Итого по выбору:

- Хотите вероятности и стабильную оптимизацию — берите `"log_loss"`.
- Максимальный зазор/линейная маржинальная постановка — `"hinge"` или `"squared_hinge"` (второй мягче по градиентам).
- Нужна робастность к выбросам и «почти вероятности» без чисто логистической модели — `"modified_huber"`.

`elasticnet`: что это за регуляризация

Смысл. *Elastic Net* объединяет L_1- и L_2-штрафы. Пусть $\alpha > 0$ — общий коэффициент регуляризации, а `l1_ratio` $\in [0, 1]$ — доля L_1. Тогда добавочный член к функции потерь имеет вид

$$\alpha\Big(\texttt{l1\_ratio}\,\|w\|_1 \;+\; (1 - \texttt{l1\_ratio})\,\tfrac{1}{2}\,\|w\|_2^2\Big).$$

L_1 продвигает *разреженность* (зануляет части весов, тем самым делает отбор признаков), а L_2 *усаживает* веса и снижает переобучение на скоррелированных признаках. Совместно они:

- лучше работают при сильной мультиколлинеарности (признаки сильно коррелируют ,то есть один можно выразить через другой) (по сравнению с чистым L_1);
- могут давать компактные и устойчивые модели (часть коэффициентов ровно нулевые, оставшиеся — «усажены»);
- требуют подбора обеих ручек: α (общая сила) и `l1_ratio` (баланс L_1/L_2).

Практика подбора.

- Начните с $\alpha \in \{10^{-4}, 10^{-3}, 10^{-2}\}$, `l1_ratio` $\in \{0.1, 0.5, 0.9\}$; уточняйте поиском по сетке.
- Если нужна сильная разреженность/отбор признаков — повышайте `l1_ratio` к 1 (вплоть до чистого L_1).
- Если признаки сильно скоррелированы и важна устойчивость — понижайте `l1_ratio` к 0 (в сторону L_2).

Мини-пример (логистическая классификация с вероятностями).

```python
from sklearn.linear_model import SGDClassifier
import numpy as np

sgd = SGDClassifier(
    loss="log_loss",
    penalty="l2",
    alpha=1e-4,
    learning_rate="optimal",
    max_iter=5000,
    tol=1e-4,
    random_state=7
).fit(Xtr, ytr)

# Вероятности (для log_loss и modified_huber доступны proba)
proba = sgd.predict_proba(Xte)[:, 1]      # P(y=1 | x)

# Альтернатива: через decision_function + сигмоида (бинарный случай)
sigmoid = lambda z: 1.0 / (1.0 + np.exp(-z))
p_alt = sigmoid(sgd.decision_function(Xte))
```

Что возвращают методы.

- `predict(X)` — метки классов формы $(N,)$.
- `predict_proba(X)` — вероятности (N, K) при `loss="log_loss"` (исп. сигмоиду) и частично `"modified_huber"`.
- `predict_log_proba(X)` — лог-вероятности (для `loss="log_loss"`).
- `decision_function(X)` — отступы: $(N,)$ в бинарном и (N, K) в многоклассовом случае. То есть сырое линейное значение до применения *сигмоиды или softmax*.
- `partial_fit(X, y[, classes])` — инкрементальное дообучение на потоках.
- `get_params()`, `set_params()` — доступ/настройка гиперпараметров.
- `densify()`, `sparsify()` — перевод `coef_` в плотный/разреженный формат (в новых версиях могут считаться устаревшими).

Полезные атрибуты.

- `coef_` (K, d), `intercept_` $(K,)$, `classes_`, `n_iter_`, `t_`, `n_features_in_`, `feature_names_in_`.

3.3 `SGDRegressor`: линейная регрессия с SGD

Назначение. Линейная регрессия (комбинация признаков, весов и байеса), обучаемая стохастическим/минибатч-градиентным спуском, с поддержкой робастных и ε-инсensitive потерь.

Ключевые параметры (с допустимыми значениями).

- `loss`: `"squared_error"` (MSE), `"huber"`, `"epsilon_insensitive"`, `"squared_epsilon_insensiti`

- `penalty` ∈ {`"l2"`, `"l1"`, `"elasticnet"`, `None`}; `alpha`; `l1_ratio`.
- `epsilon` — ширина «мёртвой зоны» для `epsilon_*` и `huber`.

- `learning_rate` ∈ {`"optimal"`, `"constant"`, `"invscaling"`, `"adaptive"`}, а также `eta0` *(learning_rate)*, *power_t.max_iter*, *tol*, *shuffle*, *random_state*, *fit_intercept*, *early_stopping*, *valid*

3.4 Функции потерь в `SGDRegressor`: `huber`, `epsilon_insensitive`, `squared_epsilon_insensitiv`

Обозначим остаток (ошибку) $r = y - \hat{y}$. Параметр `epsilon` > 0 управляет «мёртвой зоной» (нечувствительностью к малым ошибкам) для трёх функций потерь ниже.

`huber` — **робастная квадратно-линейная потеря.** Квадратичная около нуля и линейная на больших остатках:

$$\ell_{\text{Huber}}(r) = \begin{cases} \dfrac{1}{2}\, r^2, & |r| \le \epsilon, \\ \epsilon\Big(|r| - \dfrac{\epsilon}{2}\Big), & |r| > \epsilon. \end{cases}$$

Интуиция: мелкие ошибки наказываются как при MSE (гладкие градиенты), крупные — лишь линейно, что снижает влияние выбросов. Градиент по r: r при $|r| \le \epsilon$ и $\epsilon\,\text{sign}(r)$ при $|r| > \epsilon$.

`epsilon_insensitive` — **ϵ-нечувствительная потеря (как в ϵ-SVR).** Игнорирует ошибки внутри «трубы» ширины ϵ и линейно штрафует выход за неё:

$$\ell_\epsilon(r) = \max\big(0,\ |r| - \epsilon\big).$$

Подходит, когда хочется нулевого штрафа на уровне шума и линейного наказания за превышение.

`squared_epsilon_insensitive` — **квадратичная версия ϵ-нечувствительной.** Сохраняет «трубу» ширины ϵ, но за её пределами штраф квадратичный:

$$\ell_{\epsilon^2}(r) = \big(\max(0,\ |r| - \epsilon)\big)^2.$$

Дает нулевой штраф внутри шума и более сильное наказание за большие промахи (по сравнению с линейной версией).

Практические заметки.

- **Роль ϵ.** Чем больше ϵ, тем шире зона нечувствительности (или квадратичная зона в `huber`) и тем ниже чувствительность к шуму; слишком большое ϵ может вести к недообучению.
- **Когда что выбирать.** `huber` хорошо работает при выбросах; `epsilon_insensitive` и `squared_epsilon_insensitive` воспроизводят поведение ϵ-SVR в стохастической линейной постановке.
- **Регуляризация.** В `SGDRegressor` дополнительно задаются `penalty` ∈ {`l2`, `l1`, `elasticnet`} и α — они существенно влияют на устойчивость и обобщение.

Мини-пример (MSE-регрессия).

```python
from sklearn.linear_model import SGDRegressor

reg = SGDRegressor(
    loss="squared_error",
    penalty="elasticnet",
    alpha=1e-4, l1_ratio=0.15,
    learning_rate="invscaling", eta0=1e-2, power_t=0.5,
    max_iter=5000, tol=1e-4, random_state=7
).fit(Xtr, ytr)

y_pred = reg.predict(Xte)       # (N,)
r2 = reg.score(Xte, yte)        # R^2 на месте
```

Что возвращают методы.

- `predict(X)` — предсказанные значения $(N,)$.
- `score(X, y)` — R^2 по умолчанию.
- `partial_fit(X, y)` — потоковое дообучение.
- `get_params()`, `set_params()` — чтение/задание гиперпараметров.
- `coef`$^{-}$, `intercept`$^{-}$$bias()$`set_params()`$^{-}$/.

3.5 Практика и советы

- **Масштабируйте признаки** (`StandardScaler`) — критично для скорости и стабильности шага.
- Для классификации `loss="log_loss"` даёт `predict_proba`; для `hinge` вероятностей нет — используйте `decision_function`.
- Регуляризацию усиливают так: ↑`alpha` (в `SGD*`); для `elasticnet` регулируйте `l1_ratio`.
- Если обучение «дрожит», уменьшите `eta0` или выберите `learning_rate="adaptive"`.
- Для потоков данных используйте `partial_fit` и `average=True`.

3.6 Ещё короткие примеры

Линейный SVM через `SGDClassifier` (`hinge`).

```python
svm_lin = SGDClassifier(
    loss="hinge", penalty="l2", alpha=1e-4,
    learning_rate="optimal", max_iter=2000, tol=1e-4, random_state=0
).fit(Xtr, ytr)

margins = svm_lin.decision_function(Xte)  # (N,) или (N, K)
y_pred  = svm_lin.predict(Xte)
```

Робастная регрессия Huber с адаптивным шагом.

```python
hub = SGDRegressor(
    loss="huber", epsilon=0.1,
    learning_rate="adaptive", eta0=1e-2,
    penalty="l2", alpha=1e-4,
    max_iter=3000, tol=1e-4, random_state=0
).fit(Xtr, ytr)

print(hub.coef_, hub.intercept_)
```

3.7 Параметр `solver` в scikit-learn: где он есть и что означает

Сначала важная оговорка. *Не во всех* моделях scikit-learn есть явный параметр `solver`. Он присутствует лишь там, где действительно выбирается численный метод оптимизации (обычно для гладких задач). Модели наподобие деревьев решений, случайных лесов, kNN, наивного Байеса, `SVC`/`SVR` не имеют пользовательского `solver`: они используют фиксированные алгоритмы (жадное разбиение, поиск соседей, `libsvm`/SMO и т. п.).

Где `solver` есть

- `LogisticRegression`: `solver` ∈ {`lbfgs`, `newton-cg`, `sag`, `saga`, `liblinear`}. Выбор влияет на поддержку штрафов (`L1`/`L2`/`elasticnet`), мультикласс и масштаб данных.
- `Ridge`/`RidgeClassifier`: `solver` ∈ {`auto`, `svd`, `cholesky`, `lsqr`, `sparse_cg`, `sag`, `saga`}. Разные решатели для плотных/разреженных и малых/больших задач наименьших квадратов.
- `MLPClassifier`/`MLPRegressor`: `solver` ∈ {`adam`, `sgd`, `lbfgs`}. Соответственно — адаптивный стохастический метод, классический SGD или квазиньютоновский full-batch.
- `PCA`: параметр `svd_solver` ∈ {`auto`, `full`, `arpack`, `randomized`} — выбор варианта SVD/собственных значений.
- `NMF`: `solver` ∈ {`cd`, `mu`} — координатный спуск или мультипликативные обновления.

Где `solver` нет, но важны детали оптимизации

- `SVC`/`SVR`: внутри `libsvm`/SMO; `solver` не настраивается.
- `LinearSVC`/`LinearSVR`: внутри `liblinear` (варианты координатного спуска); есть `dual=True/False` для выбора прямой/двойственной постановки.
- `Lasso`/`ElasticNet`: используются варианты координатного спуска; явного `solver` нет (есть `selection`, `max_iter`, `tol` и т. п.).
- `SGDClassifier`/`SGDRegressor`: это сами по себе стохастические решатели; вместо `solver` выбираются `loss`, `penalty`, `alpha`, `learning_rate` и т. д.
- **Деревья/лес/градиентный бустинг**: нет `solver`; обучение — жадное построение/свертка деревьев (параметры: `criterion`, `max_depth`, `learning_rate` и пр.).

Карта основных оптимизаторов и как они работают

- **L-BFGS** (квазиньютоновский, full-batch). Аппроксимирует гессиан через ограниченную историю градиентов; быстрые точные шаги на гладких задачах; чувствителен к масштабированию признаков.
- **Newton-CG** (усечённый Ньютон). Использует информацию о гессиане (или его умножениях) и метод сопряжённых градиентов для внутреннего решения; хорошо для гладких задач среднего размера.
- `liblinear`. Библиотека для линейных моделей (логистическая регрессия, линейный SVM): координатный спуск/двойственный оптимизатор, эффективен на малых/-средних данных, поддерживает `L1`.
- **SAG/SAGA**. Стохастические методы со снижением дисперсии: хранят усреднённый градиент по всем объектам. `SAGA` добавляет проксимальный шаг и поддерживает разреживающие штрафы (`L1`, `elasticnet`). Требуют *обязательного* масштабирования признаков и подходят для больших датасетов.
- **SGD**. Базовый стохастический градиентный спуск с различными расписаниями шага; даёт максимальную гибкость, но требует аккуратной настройки `learning_rate`/`eta0`/моментума.
- **Adam**. Адаптивный метод первого порядка (накопление первых/вторых моментов градиента); устойчив на «шумных» ландшафтах, хорошо работает по умолчанию в `MLP`.
- **Cholesky / SVD / LSQR** / `sparse_cg` (для регрессии на МНК). Прямые (`cholesky`, `svd`) и итерационные (`lsqr`, `sparse_cg`) решатели для линейных систем; выбор зависит от размера и разреженности матрицы признаков.
- **ARPACK / randomized SVD** (в `PCA`). Итерационные методы для нескольких главных компонент (`arpack`) и стохастический быстрый SVD (`randomized`) на больших плотных данных.
- **Coordinate Descent (CD)**. Последовательная оптимизация по координатам; используется в `Lasso`/`ElasticNet` и `NMF` (`solver="cd"`); хорошо работает с разреживающими штрафами.
- **Multiplicative Updates (MU)** (`NMF`). Простые неотрицательные обновления; обычно медленнее CD, но очень стабильны при неотрицательных ограничениях.

Практические подсказки

- Для `sag`/`saga`/`sgd`/`adam` всегда масштабируйте признаки (`StandardScaler`), иначе возможны проблемы со сходимостью.
- Для логистической регрессии: `lbfgs` — надёжный дефолт; `saga` — если нужны `L1`/`elasticnet` или очень большие данные.
- Для гребневой регрессии: `svd`/`cholesky` — малые плотные; `lsqr`/`sparse_cg` — крупные/разреженные; `sag`/`saga` — очень большие.
- Деревья, леса, бустинг, kNN, `SVC` — `solver` не выбирается; на качество влияют другие гиперпараметры.

4 KNN: классифицирующие и регрессионные варианты

Примечание про теорию. Полная теория вынесена в отдельный файл. Здесь — практическая выжимка по коду и настройкам.

1) KNN-классификатор (`KNeighborsClassifier`)

Идея. Для каждого объекта находим k ближайших обучающих точек и голосуем за класс (равномерно или с весами, зависящими от расстояния). Основные гиперпараметры: `n_neighbors` (k), `weights` (`"uniform"` или `"distance"`), `metric` (по умолчанию `"minkowski"` с параметром `p`: p=2 — евклидова, p=1 — манхэттенская), `algorithm` (`"auto"`, `"kd_tree"`, `"ball_tree"`, `"brute"`). Масштабирование признаков почти всегда полезно.

```python
from sklearn.neighbors import KNeighborsClassifier
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler

# KNN-классификатор с весами по расстоянию
knn_clf = make_pipeline(
    StandardScaler(),
    KNeighborsClassifier(
        n_neighbors=5,
        weights="distance",        # или "uniform"
        metric="minkowski",
        p=2                        # 2=Euclidean, 1=Manhattan
    )
)
knn_clf.fit(Xtr, ytr)
y_pred = knn_clf.predict(Xte)
proba  = knn_clf.predict_proba(Xte)  # вероятности классов (если поддерживается)
```

Как работает.

Функция *make_pipline* позволяет собрать контейнер и последовательно применить трансформацию к данным. Стоит отметить, что у пайплайна есть атрибут `.named_steps`. Это просто словарь, в котором ключи — это имена шагов пайплайна (те строки, которые вы указали: "scaler "kneighborsclassifier"), а значения — соответствующие объекты (например, StandardScaler(), KNeighborsClassifier()). В данном случае стандартизацию и модель. Соответсвенно обращение к функциям модель можно производить через функцию `.named_steps`.

Алгоритм: *Поиск соседей → взвешенное голосование → класс/вероятности.* При `weights="distance"` ближние соседи влияют сильнее. Выбор k: малый k ⇒ риск переобучения, большой k ⇒ сглаживание границы.

4.1 Взвешенный KNN: `weights="distance"` и свои ядра

Смысл. Вместо простого голосования по k соседям используем веса, зависящие от расстояния: ближние соседи влияют сильнее, дальние — слабее. В `scikit-learn` параметр `weights` у `KNeighborsClassifier`/`KNeighborsRegressor` может быть:

- `"uniform"` — равные веса (классический KNN).
- `"distance"` — веса обратно пропорциональны расстоянию, $w_i \propto 1/d_i$; если есть нулевые расстояния, решение строится только по таким соседям (внутренняя логика `sklearn`).
- *callable* — своя функция весов $w_i = f(d_i)$.

Как это считается. Для классификации — взвешенное голосование:

$$\hat{y} = \arg\max_c \sum_{i \in \mathcal{N}_k(x)} w_i \, \mathbf{1}\{y_i = c\}.$$

Для регрессии — взвешенное среднее:

$$\hat{y} = \frac{\sum_{i \in \mathcal{N}_k(x)} w_i \, y_i}{\sum_{i \in \mathcal{N}_k(x)} w_i}.$$

Пример: `weights="distance"` (классификация).

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler

knn_w = make_pipeline(
    StandardScaler(),
    KNeighborsClassifier(n_neighbors=7, weights="distance", metric="minkowski",
    p=2)
)
knn_w.fit(Xtr, ytr)
y_pred = knn_w.predict(Xte)
proba  = knn_w.predict_proba(Xte)
```

Подводные камни.

- Масштаб признаков критичен (обязательно `StandardScaler`).
- При $d_i \approx 0$ веса могут резко «взрываться»; используйте устойчивые формулы (например, $1/(d_i + \varepsilon)$ или гауссово ядро).
- Выбор k и формы весов — по кросс-валидации; малый k переобучает, большой k — переусредняет.

2) Классификатор ближайшего центроида (`NearestCentroid`)

Идея. Для каждого класса вычисляется один центроид (среднее по признакам). Новый объект относим к классу ближайшего центроида. Это очень быстрый и интерпретируемый

базовый метод.

Гиперпараметры. `metric` (`"euclidean"` по умолчанию, можно `"manhattan"` и др.), `shrink_threshold` (опц. «усадка» центроидов для борьбы с шумом; `None` — без усадки).

```python
from sklearn.neighbors import NearestCentroid
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import pairwise_distances
import numpy as np

nc = make_pipeline(
    StandardScaler(),
    NearestCentroid(metric="euclidean", shrink_threshold=None)
)
nc.fit(Xtr, ytr)
y_pred = nc.predict(Xte)

# Уверенность: расстояния до центроидов (чем ближе, тем увереннее)

# У пайплайна nc будет атрибут .named_steps.
# Это просто словарь, в котором ключи -- это имена шагов пайплайна (те строки, к
    оторые вы указали: "scaler", "nearestcentroid"), а значения -- соответствующи
    е объекты (например, StandardScaler(), NearestCentroid()).

centroids = nc.named_steps["nearestcentroid"].centroids_
```

Как работает. *На обучении:* средние по классам → центроиды. *На предсказании:* расстояние до каждого центроида → выбираем ближайший.

3) KNN-регрессор (`KNeighborsRegressor`)

Идея. Для точки берём k ближайших *вещественных* ответов и возвращаем средневзвешенное (или простое среднее). Те же метрики и алгоритмы поиска, что и в классификации.

```python
from sklearn.neighbors import KNeighborsRegressor
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler

knn_reg = make_pipeline(
    StandardScaler(),
    KNeighborsRegressor(
        n_neighbors=7,
        weights="distance",        # ближние соседи сильнее
        metric="minkowski",
        p=2
    )
)
knn_reg.fit(Xtr, ytr)
y_hat = knn_reg.predict(Xte)
```

Как работает. *Поиск соседей → агрегация ответов*: при `weights="uniform"` это среднее по k соседям; при `"distance"` — взвешенное среднее с весами, убывающими по расстоянию. Регуляризация фактически задаётся выбором k: больше $k \Rightarrow$ сильнее сглаживание.

Практика и советы.

- Масштабируйте признаки (`StandardScaler`) перед KNN/центроидными методами — метрика чувствительна к масштабу.
- Подбирайте `n_neighbors` и `metric`/`p` по кросс-валидации. Для категориальных/смешанных признаков рассмотрите специализированные метрики.
- Для больших данных смотрите на `algorithm="kd_tree"/"ball_tree"` или используйте подвыборку/approximate NN.

4.2 K-Means (`sklearn.cluster.KMeans`)

Идея и когда применять. K-Means — классический алгоритм кластеризации: он находит K центроидов $\{c_k\}_{k=1}^{K}$ и присваивает каждой точке ближайший центроид по евклидовому расстоянию, минимизируя сумму внутрикластерных квадратов расстояний (WCSS). Полезен как быстрый базовый метод при примерно сферических, схожих по размеру кластерах и когда масштаб признаков сопоставим (желательно предварительно стандартизовать).

Ключевые параметры.

- `n_clusters` — число кластеров K (обязательный гиперпараметр).
- `init` — инициализация центроидов: `"k-means++"` (дефолт и рекомендовано) или `"random"`; можно передать пользовательские центры как массив формы (K, d).
- `n_init` — сколько раз запускать алгоритм с разной инициализацией и выбрать лучшее решение по `inertia_`; обычно ≥ 10.
- `max_iter` — максимум итераций одного запуска (`lloyd/elkan`).
- `tol` — критерий сходимости для ранней остановки.
- `algorithm` — `"lloyd"` (классический) или `"elkan"` (ускорение треугольным неравенством для евклидовой метрики).
- `random_state` — воспроизводимость инициализации.

Основные атрибуты/методы.

- `cluster_centers_` $\in \mathbb{R}^{K \times d}$ — найденные центры.
- `labels_` $\in \{0, \ldots, K-1\}^N$ — метки кластеров обучающей выборки.
- `inertia_` — итоговая сумма $\sum\_b \min\_k \|x_b - c_k\|^2$ (чем меньше, тем плотнее кластеры).
- `fit(X)`, `predict(X)`, `fit_predict(X)`.

Мини-пример (базовый сценарий).

```python
from sklearn.cluster import KMeans
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

km = KMeans(
    n_clusters=3,
    init="k-means++",
    n_init=10,
    max_iter=300,
    tol=1e-4,
    algorithm="lloyd",
    random_state=7
)
labels = km.fit_predict(X_scaled)

print("inertia:", km.inertia_)
print("centers shape:", km.cluster_centers_.shape)
```

На больших данных: `MiniBatchKMeans`. Для потоковой/большой выборки используйте мини-батчи: ускорение за счёт стохастических обновлений, возможна чуть худшая `inertia_`.

```python
from sklearn.cluster import MiniBatchKMeans

mbk = MiniBatchKMeans(
    n_clusters=3,
    batch_size=1024,
    init="k-means++",
    n_init=10,
    max_iter=100,
    random_state=7
)
labels_mb = mbk.fit_predict(X_scaled)
```

Практические советы.

- **Масштабируйте** признаки (`StandardScaler`) — без этого доминирующие масштабы смещают центры.
- Начинайте с `init="k-means++"` и `n_init`$\geq$`10`; оценивать разумный K удобно по силуэту или «локтю» `inertia_`(K).
- `algorithm="elkan"` ускоряет для плотных данных с евклидовой метрикой.

Чем K-Means отличается от «K Centroids» (Nearest Centroid)

Идеологическое отличие.

- **K-Means (unsupervised).** Игнорирует метки классов и *сам* ищет K центров, минимизируя суммарный разброс внутри кластеров. Результат — кластеры безотносительно истинных классов.

- **Nearest Centroid (supervised).** Для каждого *класса* берёт *среднее* всех его обучающих объектов (один центроид на класс), затем относит новую точку к ближайшему классовому центру. В `sklearn` это `sklearn.neighbors.NearestCentroid` (опц. `shrink_threshold` для усадки центроидов).

Последствия на практике.

- K-Means может выделить кластеры, не совпадающие с истинными классами (и их число K задаёте вы). Он полезен для разведки структуры данных и предобработки (инициализация центров, сжатие, прототипы).
- Nearest Centroid — это *классификатор*: число центров равно числу классов; обучение — просто усреднение по метке, без итеративной оптимизации.

Отличие от Soft K-Means (температурные присваивания)

Жёсткие vs мягкие назначения. Классический K-Means делает *жёсткое* присваивание (one-hot): точка попадает к *одному* ближайшему центру. Soft K-Means (как в вашем ноутбуке) использует $\texttt{softmax}(-\|x-c\|^2/\tau)$ и *мягкие* вероятности принадлежности q_b, k, добавляя энтропийный член в лосс. Это помогает стабильности на старте и даёт вероятностную интерпретацию.

Типичные ошибки и быстрые проверки

- **Не масштабированы** признаки — центры «тянут» признаки с большими шкалами.
- **Слишком маленький** n_init — попадание в плохие локальные минимумы.
- **Переоценка** $inertia\_$: она падает с ростом K всегда; используйте силуэт/BIC/AIC (для смесей) или правило «локтя».
- **Неевклидова природа данных.** K-Means оптимален для евклидовой геометрии; для категориальных/других метрик рассмотрите k-modes/k-prototypes или кластеризацию по расстояниям.

5 Решающие деревья в scikit-learn: обучение, стрижка, кодирование категорий

5.1 Бинарное решающее дерево (классификация и регрессия)

Суть. Дерево рекурсивно бьёт пространство признаков на прямоугольные области по правилам вида $x_j \le t$. В листьях хранятся прогнозы: для классификации — распределения по классам, для регрессии — средние/медианы по целям. Обучение выбирает признак и порог, максимизирующие выигрыш критерия.

Ключевые параметры `DecisionTreeClassifier`**.**

- `criterion` $\in$ {`"gini"`, `"entropy"`, `"log_loss"`}: мера неопределённости в узлах.
- `splitter` $\in$ {`"best"`, `"random"`}: перебор лучшего сплита или случайный.
- `max_depth`: ограничение глубины (целое) — пред-стрижка.
- `min_samples_split`: минимум объектов, чтобы делить узел (целое или доля $(0, 1]$).
- `min_samples_leaf`: минимум объектов в листе (целое или доля).
- `max_features` $\in$ {None, целое, доля, `"sqrt"`, `"log2"`}: сколько признаков рассматривать при поиске сплита.
- `class_weight`: веса классов (`"balanced"` или dict).
- `random_state`: воспроизводимость.
- `ccp_alpha` ≥ 0: сила пост-стрижки по минимальной сложности (Cost-Complexity Pruning).

Ключевые параметры `DecisionTreeRegressor`**.**

- `criterion` $\in$ {`"squared_error"`, `"friedman_mse"`, `"absolute_error"`, `"poisson"`}.
- Остальные параметры аналогичны: `splitter`, `max_depth`, `min_samples_split`, `min_samples_leaf`, `max_features`, `ccp_alpha`.

Мини-пример: классификация.

```python
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

Xtr, Xte, ytr, yte = train_test_split(X, y, test_size=0.2, stratify=y,
    random_state=7)

clf = DecisionTreeClassifier(
    criterion="gini",
    max_depth=None,             # без пред-стрижки по глубине
    min_samples_split=2,
    min_samples_leaf=1,
    max_features=None,
    random_state=7
)
clf.fit(Xtr, ytr)
y_pred = clf.predict(Xte)
print("accuracy:", accuracy_score(yte, y_pred))
print("depth:", clf.get_depth(), "leaves:", clf.get_n_leaves())
```

Мини-пример: регрессия.

```python
from sklearn.tree import DecisionTreeRegressor
from sklearn.metrics import mean_squared_error

# предсказание = среднее целевых значений в этом листе.
rgr = DecisionTreeRegressor(
    criterion="squared_error",
    max_depth=None,
    min_samples_leaf=1,
    random_state=7
)
rgr.fit(Xtr, ytr_reg)
y_hat = rgr.predict(Xte)
print("RMSE:", mean_squared_error(yte_reg, y_hat, squared=False))
```

5.2 Стрижка дерева (Pruning): cost-complexity и выбор `ccp_alpha`

Идея. Пост-стрижка удаляет «дорогие» ветви, которые мало улучшают качество, минимизируя $R_\alpha(T) = R(T) + \alpha|T|$, где $R(T)$ — эмпирическая ошибка, $|T|$ — число листьев. В `sklearn` используется cost-complexity pruning с параметром `ccp_alpha`.

Пост-стрижка по минимальной сложности (cost–complexity pruning)

Что такое эмпирическая ошибка $R(T)$. Под $R(T)$ понимают «наказанность» дерева на обучающей выборке: суммарную нечистоту листьев, взвешенную по числу объектов в листах. Для классификации это сумма по всем листьям $\sum_{\ell \in \text{leaves}} n_\ell \cdot \text{impurity}_\ell$, где impurity_ℓ вычисляется выбранным критерием (`gini` или `entropy`); для регрессии — аналогично, но с `squared_error`/`absolute_error`/`poisson`. Интуитивно: чем «грязнее» листья (смешение классов, высокая дисперсия), тем выше $R(T)$.

Что означает функционал $R_\alpha(T) = R(T) + \alpha|T|$. Это баланс «качества на обучении» и «сложности» дерева:

- $R(T)$ поощряет точное подгоняние (меньше — лучше).
- $|T|$ — число листьев (сложность модели); множитель $\alpha \geq 0$ штрафует за лишние листья.
- При $\alpha = 0$ оптимально максимально разветвлённое дерево (риск переобучения). При больших α дерево агрессивно режется (риск недообучения).

Минимизируя $R_\alpha(T)$, мы ищем компромисс «смещение–дисперсия»: немного жертвуем подгонкой ради лучшей обобщаемости.

Пайплайн подбора `ccp_alpha`.

1. Получить путь α-значений через `cost_complexity_pruning_path`.
2. Обучить семейство деревьев с этими α, выбрать лучшее по CV.
3. Обучить финальную модель с найденным `ccp_alpha`.

Код: подбор `ccp_alpha` с кросс-валидацией.

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import StratifiedKFold, cross_val_score
import numpy as np

base = DecisionTreeClassifier(random_state=7)
path = base.cost_complexity_pruning_path(Xtr, ytr)
ccp_alphas = path.ccp_alphas  # возрастающая сетка

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

for alpha in ccp_alphas:
    clf_alpha = DecisionTreeClassifier(random_state=7, ccp_alpha=alpha)
    cv_score = cross_val_score(clf_alpha, Xtr, ytr, cv=cv,
    scoring="accuracy").mean()
    mean_scores.append(cv_score)

best_alpha = ccp_alphas[int(np.argmax(mean_scores))]
clf_pruned = DecisionTreeClassifier(random_state=7,
    ccp_alpha=best_alpha).fit(Xtr, ytr)
print("best ccp_alpha:", best_alpha)
print("test accuracy:", clf_pruned.score(Xte, yte))
```

Как работает `cost_complexity_pruning_path`. Вызов

```
base = DecisionTreeClassifier(random_state=7)
path = base.cost_complexity_pruning_path(Xtr, ytr)
ccp_alphas = path.ccp_alphas      # возрастающая сетка alpha
impurities = path.impurities      # соответствующие R(T) для поддеревьев
```

строит *монотонный путь поддеревьев* $T_0 \supset T_1 \supset \cdots \supset T_m$, полученных последовательным удалением «наименее выгодных» ветвей. Для каждой ступени считается *эффективный* α, при котором текущая обрезка впервые становится оптимальной:

$$\alpha_t \;=\; \frac{R(t) - R(T_t)}{|T_t| - 1},$$

где t — узел, а T_t — его поддерево. Возвращаемый массив `ccp_alphas` упорядочен неубывающе; двигаясь вправо, вы получаете всё более короткие деревья, а `impurities` — соответствующие им значения $R(T)$.

Что такое CV и зачем он здесь. CV (кросс-валидация) — это процедура оценивания качества, при которой данные делят на K непересекающихся фолдов: модель обучают на $K-1$ фолдах и валидируют на оставшемся; цикл повторяют K раз и усредняют метрику. В классификации обычно берут `StratifiedKFold` (сохраняет доли классов). Здесь CV нужен, чтобы выбрать `ccp_alpha`, дающий лучшую *валидационную* метрику (например, `accuracy`, `roc_auc`) — то есть дерево, которое лучше обобщается вне обучающей выборки.

На что влияет `ccp_alpha`.

- **Степень усечения.** Чем выше α, тем сильнее стрижка: меньше узлов, ниже глубина, проще правила.
- **Смещение–дисперсия.** Небольшая α снижает смещение, но может взвинтить дисперсию (переобучение); большая α — наоборот.
- **Интерпретируемость и скорость.** Бóльшая α даёт компактные деревья: быстрее предсказывают и легче объясняются.

Краткая расшифровка кода.

- `base = DecisionTreeClassifier(...)` — базовая заготовка без явной стрижки.
- `cost_complexity_pruning_path(Xtr, ytr)` — строит цепочку поддеревьев и возвращает сетку α (`ccp_alphas`) и их $R(T)$ (`impurities`).
- `ccp_alphas` — возрастающие значения штрафа; под каждое α можно обучить дерево и оценить по CV, чтобы выбрать оптимум.

Замечания.

- Пред-стрижка (`max_depth`, `min_samples_leaf`, ...) и пост-стрижка (`ccp_alpha`) взаимодополняемы.
- Слишком большое `ccp_alpha` приведёт к сильной усечённости дерева и недообучению.

5.3 Категориальные признаки: label/one-hot/target encoding

Label/Ordinal Encoding (порядковые коды)

Смысл. Каждой категории присваивается целочисленный код. Подходит только для *порядковых* категорий; для *номинальных* может вносить ложный порядок.

Код с `OrdinalEncoder`.

```
import pandas as pd
from sklearn.preprocessing import OrdinalEncoder
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import make_pipeline
from sklearn.tree import DecisionTreeClassifier

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

enc = ColumnTransformer(transformers=[
    ("cat", OrdinalEncoder(handle_unknown="use_encoded_value",
    unknown_value=-1), cat_cols),
    ("num", "passthrough", num_cols)
])

clf_ord = make_pipeline(enc, DecisionTreeClassifier(random_state=7))
clf_ord.fit(Xtr_df, ytr)
```

Что происходит пошагово.

- **Определение признаков.** `cat_cols` — имена категориальных столбцов (`"city"`, `"segment"`), `num_cols` — числовые (`"age"`, `"income"`). Предполагается, что `Xtr_df` — `pandas.DataFrame` с этими колонками; `ytr` — целевая переменная.
- **ColumnTransformer `enc`.** Собирает *колоночный препроцессинг*:
  - Блок `("cat OrdinalEncoder(...), cat_cols)` применяет `OrdinalEncoder` к `city` и `segment`, превращая категории в целые коды $\{0, 1, 2, \dots\}$.
  - Параметр `handle_unknown="use_encoded_value"` + `unknown_value=-1` гарантирует, что *невидимые при обучении* категории на валидации/проде будут закодированы как -1, вместо ошибки.
  - Блок `("num "passthrough num_cols)` пропускает численные признаки без изменений.

  На выходе получается единая числовая матрица признаков: сначала трансформированные категориальные, затем «как есть» числовые (порядок соответствует порядку блоков).
- **Пайплайн `clf_ord`.** `make_pipeline(enc, DecisionTreeClassifier(...))` связывает препроцессинг и модель. При вызове `fit`:
  1. `enc.fit_transform(Xtr_df)` обучает кодировщик на тренировочных данных и возвращает числовую матрицу.
  2. `DecisionTreeClassifier.fit(...)` строит бинарное дерево по преобразованным признакам (критерий сплита по умолчанию `gini`).

- **Почему OrdinalEncoder уместен с деревом.** Решающее дерево делает пороговые сплиты вида `feature` $\leq \theta$. Даже если категориальные коды несут искусственный порядок, дерево может разрезать их подходящим порогом. Тем не менее, при сильной многокатегориальности чаще предпочитают `OneHotEncoder`; `OrdinalEncoder` хорош как компактный и быстрый базовый вариант.
- **Поведение на неизвестных категориях.** Если в `X_test` появится новая `"city"`, `OrdinalEncoder` вернёт -1. Модель будет трактовать это как отдельное значение и сама выберет подходящий сплит (например, ≤ -0.5).
- **Инференс.** Вызовы `clf_ord.predict(X)` и `clf_ord.predict_proba(X)` автоматически выполнят тот же `enc.transform` и затем предсказание дерева. Никакого ручного преобразования признаков делать не нужно.

Полезные замечания.

- Чтобы получить имена выходных признаков после `ColumnTransformer` (для анализа важности), используйте `enc.get_feature_names_out()` (в новых версиях `sklearn`).
- Такой пайплайн совместим с `GridSearchCV`/`RandomizedSearchCV`: можно настраивать, например, `decisiontreeclassifier__max_depth`, `decisiontreeclassifier__min_samples`, а также параметры кодировщика.
- Если категорий очень много и порядок не имеет смысла, рассмотрите `OneHotEncoder(handle_unk` — он избегает навязанного порядка, но увеличивает размерность.

One-Hot Encoding (OHE)

Смысл. Каждая категория превращается в отдельный бинарный признак. Подходит для номинальных категорий; увеличивает размерность.

Код с `OneHotEncoder`.

```
from sklearn.preprocessing import OneHotEncoder
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import make_pipeline
from sklearn.tree import DecisionTreeClassifier

ohe = ColumnTransformer(transformers=[
    ("cat", OneHotEncoder(handle_unknown="ignore", sparse_output=False),
    cat_cols),
    ("num", "passthrough", num_cols)
])

clf_ohe = make_pipeline(ohe, DecisionTreeClassifier(random_state=7))
clf_ohe.fit(Xtr_df, ytr)
```

Замечания.

- `handle_unknown="ignore"` безопасно обрабатывает новые категории на тесте.

- Если sparse_output=True, то возвращается scipy.sparse разреженная матрица. Если False, то возвращается как обычный numpy.array.
- Для деревьев масштабирование не критично, но аккуратная обработка пропусков и редких категорий важна.

Target Mean Encoding (сглаженное среднее по категории)

Смысл. Категория кодируется *средним таргетом* по группе. Риск утечки! Делайте по фолдам, со сглаживанием к глобальному среднему.

Простой сглаженный target encoding (без внешних пакетов).

```python
import numpy as np
import pandas as pd
from sklearn.base import BaseEstimator, TransformerMixin

class MeanTargetEncoder(BaseEstimator, TransformerMixin):
    def __init__(self, cols, m=100):
        self.cols = cols
        self.m = m  # сила сглаживания
    def fit(self, X, y):
        Xy = pd.DataFrame(X, copy=False)
        Xy["_y_"] = y
        self.global_mean_ = float(np.mean(y))
        self.maps_ = {}
        for c in self.cols:
            grp = Xy.groupby(c)["_y_"].agg(["mean", "count"])
            smooth = (grp["count"] * grp["mean"] + self.m * self.global_mean_)
    / (grp["count"] + self.m)
            self.maps_[c] = smooth.to_dict()
        return self
    def transform(self, X):
        X = pd.DataFrame(X, copy=True)
        for c in self.cols:
            X[c] = X[c].map(self.maps_[c]).fillna(self.global_mean_)
        return X

# пример использования
te_cols = ["city", "segment"]
te = MeanTargetEncoder(cols=te_cols, m=50).fit(Xtr_df, ytr)
Xtr_te = te.transform(Xtr_df)
Xte_te = te.transform(Xte_df)

clf_te = DecisionTreeClassifier(random_state=7).fit(Xtr_te, ytr)
```

Как работает (кратко).

- **Идея.** Для каждой категориальной колонки c заменяем категорию на *сглаженную среднюю* таргета: чем больше наблюдений у категории, тем ближе значение к её эмпирическому среднему; редкие категории притягиваются к глобальному среднему по выборке.

- **Сглаживание.** Для категории с эмпирическими `mean` и `count` $= n$:

$$\mathrm{enc}(c) = \frac{n \cdot \texttt{mean} + m \cdot \texttt{global\_mean}}{n + m},$$

  где m — гиперпараметр сглаживания (`m` в конструкторе).
- **fit.** Сохраняет `global_mean_` $= \mathbb{E}[y]$ и словари `maps_`$[c]$: категория $\to$ сглаженная средняя для каждого столбца из `cols`.
- **transform.** Заменяет значения в `cols` на закодированные. Неизвестные/редкие категории и `NaN` получают `global_mean_` благодаря `fillna`.
- **Интеграция со sklearn.** Наследование от `BaseEstimator`, `TransformerMixin` делает трансформер совместимым с `Pipeline`, `GridSearchCV`.

Какие объекты подавать на вход.

- X: табличные признаки формы (N, d) — `pandas.DataFrame` (предпочтительно, т.к. используются имена колонок) или любой массивоподобный объект, который можно обернуть в `DataFrame`. В `cols` перечислены *существующие* в X категориальные столбцы; их значения должны быть хешируемыми (строки/числа).
- y: одномерный вектор длины N (`pandas.Series` или `numpy.ndarray`) с числовыми таргетами. Для бинарной классификации обычно $\{0, 1\}$. Для регрессии — любые вещественные.
- **Выход `transform`.** `pandas.DataFrame` той же формы, где указанные `cols` заменены на вещественные (кодированные) значения; остальные столбцы сохранены как есть.

Практические замечания.

- **Избегайте утечки.** Используйте трансформер *внутри* `Pipeline` с валидатором (`cross_val_score`, `GridSearchCV`), чтобы `fit` выполнялся только на тренировочных фолдах.
- **Подбор `m`.** Больший `m` сильнее тянет к глобальному среднему (полезно при редких категориях), меньший `m` позволяет лучше использовать частые категории.
- **Неизвестные категории.** На тесте получают `global_mean_` благодаря `fillna`; это безопаснее, чем падать с ошибкой.

Важно про утечку.

- Делайте target encoding *внутри* кросс-валидации (out-of-fold), иначе модель «подсмотрит» таргет.
- Параметр `m` регулирует сглаживание: малое $\to$ переобучение на редких категориях, большое $\to$ сильная усреднённость.

6 SVM и SVR: быстрый практический конспект

6.1 SVM для классификации (`SVC`, `LinearSVC`)

Идея. Опорные векторы строят разделяющую гиперплоскость с *максимальным зазором* (margin). В мягком варианте решается

$$\min_{w,b,\xi} \ \tfrac{1}{2}\|w\|^2 + C\sum_i \xi_i \quad \text{s.t.} \quad y_i\,(w^\top \phi(x_i) + b) \geq 1 - \xi_i,\ \xi_i \geq 0,$$

где ϕ — (возможно, неявное) отображение в признаковое пространство, реализуемое ядром $K(x, x') = \langle \phi(x), \phi(x') \rangle$. Параметр $C > 0$ контролирует баланс между шириной зазора и штрафом за ошибки.

Когда что брать.

- `SVC` ($\sim$ LIBSVM): поддерживает любые ядра, хорош для малых/средних датасетов, работает в дуале.
- `LinearSVC` ($\sim$ LIBLINEAR): линейное ядро, масштабируется на большие/разреженные данные (тексты). Быстрее на $d \gg n$.

Код: RBF–SVM с масштабированием и вероятностями.

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

svc = make_pipeline(
    StandardScaler(),
    SVC(kernel="rbf", C=1.0, gamma="scale", probability=True,
        class_weight=None, tol=1e-3, cache_size=200, max_iter=-1)
)
svc.fit(Xtr, ytr)
proba = svc.predict_proba(Xte)[:, 1]   # вероятность положительного класса
```

Код: линейный SVM для больших/разреженных данных.

```
from sklearn.svm import LinearSVC
from sklearn.calibration import CalibratedClassifierCV

lin = LinearSVC(C=1.0, loss="squared_hinge", tol=1e-3, max_iter=5000)
lin_cal = CalibratedClassifierCV(lin, method="sigmoid", cv=5)  # калибруем proba
lin_cal.fit(Xtr, ytr)
proba = lin_cal.predict_proba(Xte)[:, 1]
```

Ключевые параметры.

- `kernel`: `"linear"`, `"rbf"`, `"poly"`, `"sigmoid"`.
- `C` (> 0): сила штрафа за ошибки. Больше $C \Rightarrow$ меньше ошибок на трейне, риск переобучения.
- `tol`: порог остановки для оптимизации. Когда изменения в целевой функции становятся меньше `tol`, обучение останавливается.
- `max`$_iter = -1$.`gamma`$(RBF/Poly/Sigmoid)$: ."`scale`$= 1/(d\,\mathrm{Var}(X))$ `(по умолчанию)`, `"auto"` $= 1/d$, `или число`.
- `degree, coef0 (для poly/sigmoid): степень полинома и сдвиг.`
- `cache`$_size = 200$: .`probability=True` : `predict_proba`.
- `class_weight: "balanced" для автоматического учёта дисбаланса.`

Полезные атрибуты (для `SVC`).

- `support_` (индексы опорных векторов), `support_vectors_`, `n_support_`.
- `dual_coef_`, `intercept_` (bias). Для `kernel="linear"` доступен `coef_` (веса гиперплоскости).

6.2 SVR для регрессии (`SVR`, `LinearSVR`)

Идея. ε-инвариантная регрессия подбирает функцию $f(x) = w^\top \phi(x) + b$, которая игнорирует *мелкие* ошибки $|y_i - f(x_i)| \le \varepsilon$:

$$\min_{w,b,\xi,\xi^*} \ \tfrac{1}{2}\|w\|^2 + C\sum_i (\xi_i + \xi_i^*) \ \text{ s.t. } \begin{cases} y_i - f(x_i) \le \varepsilon + \xi_i, \\ f(x_i) - y_i \le \varepsilon + \xi_i^*, \\ \xi_i, \xi_i^* \ge 0. \end{cases}$$

Параметр ε задаёт ширину «трубки» нечувствительности, C штрафует отклонения за её пределами.

Код: RBF–SVR с масштабированием.

```
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVR

svr = make_pipeline(
    StandardScaler(),
    SVR(kernel="rbf", C=10.0, epsilon=0.1, gamma="scale", tol=1e-3,
    cache_size=200)
)
svr.fit(Xtr, ytr)
yhat = svr.predict(Xte)
```

Код: `LinearSVR` для больших/разреженных задач.

```python
from sklearn.svm import LinearSVR

lin_svr = LinearSVR(C=1.0, epsilon=0.1, loss="squared_epsilon_insensitive",
                    tol=1e-3, max_iter=5000, random_state=7)
lin_svr.fit(Xtr, ytr)
yhat = lin_svr.predict(Xte)
```

Ключевые параметры (SVR/LinearSVR).

- `kernel` (`SVR`): `"rbf"` (часто лучший базовый), `"linear"`, `"poly"`, `"sigmoid"`.
- `C` (> 0): сила штрафа за выход за «трубку». Больше $C \Rightarrow$ меньше смещение, больше риск переобучения.
- `epsilon`: ширина ε-трубки; больше $\varepsilon \Rightarrow$ модель менее чувствительна к мелким шумам.
- `gamma` (ядровые SVR): масштаб ядра (`"scale"` по умолчанию).
- `loss` (`LinearSVR`): `"epsilon_insensitive"` или `"squared_epsilon_insensitive`.

Атрибуты.

- `SVR`: `support_vectors_`, `dual_coef_`, `intercept_`; для `kernel="linear"` — `coef_`.
- `LinearSVR`: оптимизирует в *прямой* постановке, не хранит опорные векторы (экономнее по памяти/времени).

Практические советы.

- Масштабируйте признаки. Без скейлинга выбор γ, C, ε становится непредсказуемым.
- Для больших и разреженных данных пробуйте `LinearSVR`.