Курс лекций

Искусственный интеллект: от данных к решениям

Преподаватель:
Чернов Владимир Евгеньевич

Учебный год:
2025-2026

Государственное бюджетное общеобразовательное учреждение города Москвы
«Лицей «Вторая школа» имени В.Ф. Овчинникова»

Содержание

1. Понятие искусственного интеллекта. Основы линейной алгебры

1.1. Что такое искусственный интеллект?

Искусственный интеллект (ИИ) — это область компьютерной науки, занимающаяся созданием систем, способных выполнять задачи, требующие человеческого интеллекта.

Машинное обучение - это подраздел искусственного интеллекта, который занимается разработкой алгоритмов, которые могут обучаться на основе данных и опыта. С помощью машинного обучения компьютерные системы могут самостоятельно улучшать свою производительност и адаптироваться к новым задачам.

Глубокое обучение — это подраздел машинного обучения, использующий искусственные нейронные сети для эмуляциии работы человеческого мозга.

1.2. Задачи машинного и глубокого обучения

Несколько примеров задач, которые может решать искусственный интеллект:

- Анализ больших объёмов данных
- Распознавание образов и объектов
- Обработка естественного языка и автоматический перевод
- Разработка автономных транспортных средств

Когда Netflix рекомендует вам фильм — это машинное обучение. Когда телефон разблокируется по лицу — это компьютерное зрение. Когда переводчик переводит текст — это обработка естественного языка.

1.3. Векторы и матрицы

Определение 1.1. Вектор

Вектор — это упорядоченный набор чисел, который можно представить как:

$$v = \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{pmatrix} \text{(вектор-столбец)} \quad \text{или} \quad v = (v_1, v_2, \dots, v_n) \text{(вектор-строка)}$$

где v_i — компоненты вектора.

Определение 1.2. Матрица

Матрица - это прямоугольная таблица чисел размера $n \times m$ (n строк, m столбцов):

$$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1m} \\ a_{21} & a_{22} & \cdots & a_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nm} \end{pmatrix}$$

Пример вектора признаков объекта (данные ириса Фишера):

$$x = (5.1 \quad 3.5 \quad 1.4 \quad 0.2)$$

Матрица данных для набора объектов:

$$X = \begin{pmatrix} x_{11} & x_{12} & x_{13} & x_{14} \\ x_{21} & x_{22} & x_{23} & x_{24} \\ \vdots & \vdots & \vdots & \vdots \\ x_{n1} & x_{n2} & x_{n3} & x_{n4} \end{pmatrix}$$

Каждая строка этой матрицы является вектором данных для каждого из объектов.

1.4. Основные операции

Определение 1.3. Сложение векторов и матриц

Сложение векторов или матриц выполняется по-элементно, а следовательно данная операция применима только при равных размерностях векторов или же матриц.

Пример сложения векторов:

$$c = a + b = \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{pmatrix} + \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix} = \begin{pmatrix} a_1 + b_1 \\ a_2 + b_2 \\ \vdots \\ a_n + b_n \end{pmatrix}$$

Пример сложения матриц:

$$C = A + B = a_{i,j} + b_{i,j} \; \forall i, j = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1m} \\ a_{21} & a_{22} & \cdots & a_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nm} \end{pmatrix} + \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1m} \\ b_{21} & b_{22} & \cdots & b_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{nm} \end{pmatrix} =$$

$$= \begin{pmatrix} a_{11} + b_{11} & a_{12} + b_{12} & \cdots & a_{1m} + b_{1m} \\ a_{21} + b_{21} & a_{22} + b_{22} & \cdots & a_{2m} + b_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} + b_{n1} & a_{n2} + b_{n2} & \cdots & a_{nm} + b_{nm} \end{pmatrix}$$

Определение 1.4. Умножение матрицы на скаляр

Если A — матрица размера $n \times m$ и λ — скаляр (число), то их произведение λA — матрица размера $n \times m$, каждый элемент которой получен умножением соответствующего элемента матрицы A на скаляр λ:

$$\lambda A = \lambda \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1m} \\ a_{21} & a_{22} & \cdots & a_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nm} \end{pmatrix}_{n \times m} = \begin{pmatrix} \lambda a_{11} & \lambda a_{12} & \cdots & \lambda a_{1m} \\ \lambda a_{21} & \lambda a_{22} & \cdots & \lambda a_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ \lambda a_{n1} & \lambda a_{n2} & \cdots & \lambda a_{nm} \end{pmatrix}_{n \times m}$$

где $(\lambda A)_{ij} = \lambda a_{ij} \; \forall i, j$.

При умножении матрицы на скаляр каждый элемент матрицы умножается на это число, сохраняя размерность исходной матрицы.

Пример умножения матрицы на скаляр:

$$2 \cdot \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}_{2 \times 3} = \begin{pmatrix} 2 \cdot 1 & 2 \cdot 2 & 2 \cdot 3 \\ 2 \cdot 4 & 2 \cdot 5 & 2 \cdot 6 \end{pmatrix} = \begin{pmatrix} 2 & 4 & 6 \\ 8 & 10 & 12 \end{pmatrix}_{2 \times 3}$$

Теорема 1.1. Свойства умножения на скаляр

- $\lambda(\mu A) = (\lambda\mu)A$
- $(\lambda + \mu)A = \lambda A + \mu A$
- $\lambda(A + B) = \lambda A + \lambda B$
- $1 \cdot A = A$
- $0 \cdot A = 0$, 0 - нулевая матрица
- $(-1) \cdot A = -A$

Определение 1.5. Умножение матриц

Если A — матрица размера $n \times m$ и B — матрица размера $m \times p$, то их произведение C — матрица размера $n \times p$:

$$C = AB = \begin{pmatrix} a_{11} & \cdots & a_{1m} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nm} \end{pmatrix} \begin{pmatrix} b_{11} & \cdots & b_{1p} \\ \vdots & \ddots & \vdots \\ b_{m1} & \cdots & b_{mp} \end{pmatrix} = \begin{pmatrix} c_{11} & \cdots & c_{1p} \\ \vdots & \ddots & \vdots \\ c_{n1} & \cdots & c_{np} \end{pmatrix}$$

где $c_{ij} = \sum_{k=1}^{m} a_{ik} b_{kj} \ \forall i, j.$

Матрицы можно умножать тогда и только тогда, когда число столбцов первой матрицы совпадает с числом строк второй.

Пример умножения матриц:

$$\underset{3\times2}{\begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix}} \underset{2\times3}{\begin{pmatrix} 7 & 8 & 9 \\ 10 & 11 & 12 \end{pmatrix}} = \begin{pmatrix} 1\cdot7+2\cdot10 & 1\cdot8+2\cdot11 & 1\cdot9+2\cdot12 \\ 3\cdot7+4\cdot10 & 3\cdot8+4\cdot11 & 3\cdot9+4\cdot12 \\ 5\cdot7+6\cdot10 & 5\cdot8+6\cdot11 & 5\cdot9+6\cdot12 \end{pmatrix} = \underset{3\times3}{\begin{pmatrix} 27 & 30 & 33 \\ 61 & 68 & 75 \\ 95 & 106 & 117 \end{pmatrix}}$$

Определение 1.6. Транспонирование матрицы

Если A — матрица размера $n \times m$, то её транспонированная матрица A^T — матрица размера $m \times n$, полученная заменой строк на столбцы:

$$A = \underset{n\times m}{\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1m} \\ a_{21} & a_{22} & \cdots & a_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nm} \end{pmatrix}} \quad \Rightarrow \quad A^T = \underset{m\times n}{\begin{pmatrix} a_{11} & a_{21} & \cdots & a_{n1} \\ a_{12} & a_{22} & \cdots & a_{n2} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1m} & a_{2m} & \cdots & a_{nm} \end{pmatrix}}$$

где $(A^T)_{ij} = a_{ji} \ \forall i, j.$

Элемент, стоящий в i-й строке и j-м столбце исходной матрицы, переходит в j-ю строку и i-й столбец транспонированной матрицы.

Пример транспонирования матрицы:

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \end{pmatrix}^T_{4\times 3} = \begin{pmatrix} 1 & 4 & 7 & 10 \\ 2 & 5 & 8 & 11 \\ 3 & 6 & 9 & 12 \end{pmatrix}_{3\times 4}$$

Свойство 1.1. Свойства операции транспонирования

- $(A^T)^T = A$
- $(A + B)^T = A^T + B^T$
- $(AB)^T = B^T A^T$
- $(\lambda A)^T = \lambda A^T$, $\lambda \in \mathbb{R}$

Определение 1.7. Симметричная матрица

Матрица A называется симметричной, если $A^T = A$. Все симметричные матрицы квадратные и их элементы симметричны относительно главной диагонали.

Пример симметричной матрицы:

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 2 & 4 & 5 \\ 3 & 5 & 6 \end{pmatrix} \quad \text{и} \quad A^T = \begin{pmatrix} 1 & 2 & 3 \\ 2 & 4 & 5 \\ 3 & 5 & 6 \end{pmatrix} \quad \Rightarrow \quad A = A^T$$

Определение 1.8. Длина вектора

Длина (норма) вектора $v = \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{pmatrix}$ — это число, характеризующее его "размер"в пространстве и вычисляемое по формуле:

$$\|v\| = \sqrt{v_1^2 + v_2^2 + \cdots + v_n^2}$$

Для векторов на плоскости ($\mathbb{R}^2$) и в пространстве ($\mathbb{R}^3$) длина имеет геометрический смысл расстояния от начала координат до точки, которую задаёт вектор.

Примеры вычисления длины вектора:

$$\left\| \begin{pmatrix} 3 \\ 4 \end{pmatrix} \right\| = \sqrt{3^2 + 4^2} = \sqrt{9 + 16} = \sqrt{25} = 5$$

$$\left\| \begin{pmatrix} 1 \\ 2 \\ 2 \end{pmatrix} \right\| = \sqrt{1^2 + 2^2 + 2^2} = \sqrt{1 + 4 + 4} = \sqrt{9} = 3$$

$$\left\| \begin{pmatrix} 2 \\ -1 \\ 3 \\ 1 \end{pmatrix} \right\| = \sqrt{2^2 + (-1)^2 + 3^2 + 1^2} = \sqrt{4 + 1 + 9 + 1} = \sqrt{15}$$

Свойство 1.2. Свойства длины вектора

- **Неотрицательность**: $\|v\| \geq 0$, причём $\|v\| = 0$ только если $v = 0$
- **Однородность**: $\|\lambda v\| = |\lambda| \cdot \|v\|$ для любого скаляра λ
- **Неравенство треугольника**: $\|u + v\| \leq \|u\| + \|v\|$
- **Единичный вектор**: вектор $\frac{v}{\|v\|}$ имеет длину 1 и называется нормированным

Замечание 1. Геометрическая интерпретация

В двумерном пространстве длина вектора $\begin{pmatrix} x \\ y \end{pmatrix}$ — это гипотенуза прямоугольного треугольника с катетами $|x|$ и $|y|$ (теорема Пифагора). В трёхмерном пространстве — диагональ прямоугольного параллелепипеда.

1.5. Скалярное произведение векторов

Определение 1.9. Скалярное произведение векторов

Если u и v — векторы размера $n \times 1$, то их скалярное произведение — это число, равное сумме произведений соответствующих координат:

$$u \cdot v = u_1 v_1 + u_2 v_2 + \cdots + u_n v_n = \sum_{i=1}^{n} u_i v_i$$

Скалярное произведение можно также выразить через умножение векторов:

$$u \cdot v = u^T v = \begin{pmatrix} u_1 & u_2 & \cdots & u_n \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{pmatrix}$$

Результат скалярного произведения — скаляр (число), а не вектор.

Примеры скалярного произведения:

$$\begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} \cdot \begin{pmatrix} 4 \\ 5 \\ 6 \end{pmatrix} = 1 \cdot 4 + 2 \cdot 5 + 3 \cdot 6 = 4 + 10 + 18 = 32$$

$$\begin{pmatrix} 2 \\ -1 \\ 3 \\ 0 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 4 \\ -2 \\ 5 \end{pmatrix} = 2 \cdot 1 + (-1) \cdot 4 + 3 \cdot (-2) + 0 \cdot 5 = 2 - 4 - 6 + 0 = -8$$

Свойство 1.3. Свойства скалярного произведения

- **Коммутативность**: $u \cdot v = v \cdot u$
- **Дистрибутивность**: $u \cdot (v + w) = u \cdot v + u \cdot w$
- **Ассоциативность со скаляром**: $(\lambda u) \cdot v = \lambda (u \cdot v)$

- **Связь с длиной вектора**: $u \cdot u = \|u\|^2 \geq 0$, причём $u \cdot u = 0$ только если $u = 0$

Определение 1.10. Длина вектора

Длина (норма) вектора u вычисляется через скалярное произведение:

$$\|u\| = \sqrt{u \cdot u} = \sqrt{u_1^2 + u_2^2 + \cdots + u_n^2}$$

Определение 1.11. Угол между векторами

Если u и v — ненулевые векторы, то угол α между ними определяется формулой:

$$\cos\alpha = \frac{u \cdot v}{\|u\| \cdot \|v\|}$$

Векторы перпендикулярны (ортогональны) тогда и только тогда, когда $u \cdot v = 0$.

1.6. Определитель матрицы

Определение 1.12. Определитель матрицы

Определитель (детерминант) — это числовая характеристика квадратной матрицы, которая имеет важное значение в линейной алгебре. Определитель матрицы A обозначается $\det(A)$ или $|A|$.
Для матрицы 2×2 определитель вычисляется по формуле:

$$\det \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} = a_{11}a_{22} - a_{12}a_{21}$$

Для матрицы 3×3 используется правило Саррюса:

$$\det \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} = a_{11}a_{22}a_{33}+a_{12}a_{23}a_{31}+a_{13}a_{21}a_{32}-a_{13}a_{22}a_{31}-a_{12}a_{21}a_{33}-a_{11}a_{23}a_{32}$$

Примеры вычисления определителя:

$$\det \begin{pmatrix} 2 & 1 \\ 3 & 4 \end{pmatrix} = 2 \cdot 4 - 1 \cdot 3 = 8 - 3 = 5$$

$$\det \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} = 1{\cdot}5{\cdot}9+2{\cdot}6{\cdot}7+3{\cdot}4{\cdot}8-3{\cdot}5{\cdot}7-2{\cdot}4{\cdot}9-1{\cdot}6{\cdot}8 = 45+84+96-105-72-48 = 0$$

Определение 1.13. Элементарные преобразования матрицы

Элементарные преобразования — это операции над строками или столбцами матрицы, которые не меняют её определитель либо меняют его предсказуемым образом. Эти преобразования используются для упрощения вычисления определителя.

Свойство 1.4. Влияние элементарных преобразований на определитель

- **Перестановка двух строк (столбцов)**: определитель меняет знак

$$\text{Если } A \xrightarrow{\text{swap } i \leftrightarrow j} B, \text{ то } \det(B) = -\det(A)$$

- **Умножение строки (столбца) на число** $\lambda \neq 0$: определитель умножается на λ

$$\text{Если } A \xrightarrow{\text{row}_i \to \lambda \cdot \text{row}_i} B, \text{ то } \det(B) = \lambda \det(A)$$

- **Прибавление к одной строке (столбцу) другой, умноженной на число**: определитель не меняется

$$\text{Если } A \xrightarrow{\text{row}_i \to \text{row}_i + \lambda \cdot \text{row}_j} B, \text{ то } \det(B) = \det(A)$$

Вычислим определитель матрицы $B = \begin{pmatrix} 2 & 1 & 3 \\ 1 & 4 & 2 \\ 3 & 2 & 1 \end{pmatrix}$:

Шаг 1: Исходный определитель

$$\det(B) = \det \begin{pmatrix} 2 & 1 & 3 \\ 1 & 4 & 2 \\ 3 & 2 & 1 \end{pmatrix}$$

Шаг 2: Меняем местами первую и вторую строки (определитель меняет знак)

$$\text{swap row}_1 \leftrightarrow \text{row}_2: \quad -\det \begin{pmatrix} 1 & 4 & 2 \\ 2 & 1 & 3 \\ 3 & 2 & 1 \end{pmatrix}$$

Шаг 3: Вычитаем из второй строки первую, умноженную на 2

$$\text{row}_2 \to \text{row}_2 - 2 \cdot \text{row}_1: \quad -\det \begin{pmatrix} 1 & 4 & 2 \\ 0 & -7 & -1 \\ 3 & 2 & 1 \end{pmatrix}$$

Шаг 4: Вычитаем из третьей строки первую, умноженную на 3

$$\text{row}_3 \to \text{row}_3 - 3 \cdot \text{row}_1: \quad -\det \begin{pmatrix} 1 & 4 & 2 \\ 0 & -7 & -1 \\ 0 & -10 & -5 \end{pmatrix}$$

Шаг 5: Умножаем вторую строку на -1 (определитель умножается на -1)

$$\text{row}_2 \to -1 \cdot \text{row}_2: \quad \det \begin{pmatrix} 1 & 4 & 2 \\ 0 & 7 & 1 \\ 0 & -10 & -5 \end{pmatrix}$$

Шаг 6: Прибавляем к третьей строке вторую, умноженную на $\frac{10}{7}$

$$\text{row}_3 \to \text{row}_3 + \frac{10}{7} \cdot \text{row}_2: \quad \det \begin{pmatrix} 1 & 4 & 2 \\ 0 & 7 & 1 \\ 0 & 0 & -\frac{25}{7} \end{pmatrix}$$

Шаг 7: Вычисляем определитель треугольной матрицы

$$\det \begin{pmatrix} 1 & 4 & 2 \\ 0 & 7 & 1 \\ 0 & 0 & -\frac{25}{7} \end{pmatrix} = 1 \cdot 7 \cdot \left(-\frac{25}{7}\right) = -25$$

Таким образом, $\det(B) = -25$.

Элементарные преобразования позволяют привести матрицу к треугольному виду, после чего определитель равен произведению диагональных элементов. Этот метод особенно эффективен для матриц больших размеров и позволяет избежать сложных вычислений по формулам разложения.

Определение 1.14. Миноры и алгебраические дополнения

Минор M_{ij} элемента a_{ij} — это определитель матрицы, полученной из исходной удалением i-й строки и j-го столбца.

Алгебраическое дополнение A_{ij} элемента a_{ij} вычисляется по формуле:

$$A_{ij} = (-1)^{i+j} M_{ij}$$

Определение 1.15. Разложение определителя

Определитель можно вычислить разложением по любой строке или столбцу:

Разложение по i-й строке:

$$\det(A) = \sum_{j=1}^{n} a_{ij} A_{ij} = a_{i1} A_{i1} + a_{i2} A_{i2} + \cdots + a_{in} A_{in}$$

Разложение по j-му столбцу:

$$\det(A) = \sum_{i=1}^{n} a_{ij} A_{ij} = a_{1j} A_{1j} + a_{2j} A_{2j} + \cdots + a_{nj} A_{nj}$$

Вычислим определитель матрицы $A = \begin{pmatrix} 1 & 2 & 3 \\ 0 & 4 & 5 \\ 1 & 0 & 6 \end{pmatrix}$ разложением по первой строке:

Алгебраические дополнения:

$$A_{11} = (-1)^{1+1} \det \begin{pmatrix} 4 & 5 \\ 0 & 6 \end{pmatrix} = 1 \cdot (4 \cdot 6 - 5 \cdot 0) = 24$$

$$A_{12} = (-1)^{1+2} \det \begin{pmatrix} 0 & 5 \\ 1 & 6 \end{pmatrix} = (-1) \cdot (0 \cdot 6 - 5 \cdot 1) = 5$$

$$A_{13} = (-1)^{1+3} \det \begin{pmatrix} 0 & 4 \\ 1 & 0 \end{pmatrix} = 1 \cdot (0 \cdot 0 - 4 \cdot 1) = -4$$

Определитель:

$$\det(A) = 1 \cdot 24 + 2 \cdot 5 + 3 \cdot (-4) = 24 + 10 - 12 = 22$$

Свойство 1.5. Свойства определителя

- **Определитель единичной матрицы**: $\det(E) = 1$

- **Определитель нулевой матрицы**: $\det(0) = 0$
- **Определитель треугольной матрицы**: равен произведению диагональных элементов
- **Умножение на скаляр**: $\det(\lambda A) = \lambda^n \det(A)$ для матрицы $n \times n$
- **Произведение матриц**: $\det(AB) = \det(A)\det(B)$
- **Транспонирование**: $\det(A^T) = \det(A)$
- **Обратная матрица**: если $\det(A) \neq 0$, то $\det(A^{-1}) = \frac{1}{\det(A)}$
- **Перестановка строк**: при перестановке двух строк определитель меняет знак
- **Пропорциональные строки**: если две строки пропорциональны, то $\det(A) = 0$

1.7. Виды матриц и их свойства

Определение 1.16. Виды матриц

Нулевая матрица — матрица, все элементы которой равны нулю:

$$0 = \begin{pmatrix} 0 & 0 & \cdots & 0 \\ 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 \end{pmatrix}_{n \times m}$$ Обозначение: $0_{n \times m}$

Единичная матрица — квадратная матрица, у которой на главной диагонали стоят единицы, а остальные элементы равны нулю:

$$E = \begin{pmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{pmatrix}_{n \times n}$$ Обозначение: E_n

Диагональная матрица — квадратная матрица, у которой все элементы вне главной диагонали равны нулю:

$$D = \begin{pmatrix} d_{11} & 0 & \cdots & 0 \\ 0 & d_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & d_{nn} \end{pmatrix}_{n \times n}$$ Обозначение: $\text{diag}(d_{11}, d_{22}, \ldots, d_{nn})$

Верхняя треугольная матрица — квадратная матрица, у которой все элементы ниже главной диагонали равны нулю:

$$U = \begin{pmatrix} u_{11} & u_{12} & \cdots & u_{1n} \\ 0 & u_{22} & \cdots & u_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & u_{nn} \end{pmatrix}_{n \times n}$$

Нижняя треугольная матрица — квадратная матрица, у которой все элементы выше главной диагонали равны нулю:

$$L = \begin{pmatrix} l_{11} & 0 & \cdots & 0 \\ l_{21} & l_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ l_{n1} & l_{n2} & \cdots & l_{nn} \end{pmatrix}_{n \times n}$$

Примеры различных видов матриц:

Нулевая: $0_{2\times 3} = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$

Единичная: $E_3 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$

Диагональная: $\text{diag}(2,5,1) = \begin{pmatrix} 2 & 0 & 0 \\ 0 & 5 & 0 \\ 0 & 0 & 1 \end{pmatrix}$

Верхняя треугольная: $\begin{pmatrix} 1 & 2 & 3 \\ 0 & 4 & 5 \\ 0 & 0 & 6 \end{pmatrix}$

Нижняя треугольная: $\begin{pmatrix} 1 & 0 & 0 \\ 2 & 3 & 0 \\ 4 & 5 & 6 \end{pmatrix}$

Свойство 1.6. Свойства специальных матриц

- **Единичная матрица**: $AE = EA = A$ для любой квадратной матрицы A
- **Нулевая матрица**: $A + 0 = 0 + A = A$, $A \cdot 0 = 0 \cdot A = 0$
- **Диагональные матрицы**: произведение диагональных матриц коммутативно
- **Треугольные матрицы**:
  - Произведение верхних треугольных матриц — верхняя треугольная
  - Произведение нижних треугольных матриц — нижняя треугольная
  - Определитель треугольной матрицы равен произведению диагональных элементов

Пример умножения матрицы на единичную матрицу:

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \quad \text{и} \quad \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

1.8. Обратная матрица

Определение 1.17. Обратная матрица

Матрица A^{-1} называется обратной к квадратной матрице A, если выполняется условие:

$$AA^{-1} = A^{-1}A = E$$

где E — единичная матрица. Матрица, имеющая обратную, называется обратимой или невырожденной.

Теорема 1.2. Условие обратимости

Матрица A обратима тогда и только тогда, когда $\det(A) \neq 0$. Если $\det(A) = 0$, то матрица называется вырожденной и не имеет обратной.

1.8.1 Метод присоединённой матрицы

Определение 1.18. Присоединённая матрица

Присоединённая (союзная) матрица $\text{adj}(A)$ — это транспонированная матрица алгебраических дополнений:

$$\text{adj}(A) = \begin{pmatrix} A_{11} & A_{21} & \cdots & A_{n1} \\ A_{12} & A_{22} & \cdots & A_{n2} \\ \vdots & \vdots & \ddots & \vdots \\ A_{1n} & A_{2n} & \cdots & A_{nn} \end{pmatrix}$$

где A_{ij} — алгебраическое дополнение элемента a_{ij}.

Теорема 1.3. Формула обратной матрицы

Если $\det(A) \neq 0$, то обратная матрица вычисляется по формуле:

$$A^{-1} = \frac{1}{\det(A)} \text{adj}(A)$$

Найдём обратную матрицу для $A = \begin{pmatrix} 2 & 1 \\ 1 & 3 \end{pmatrix}$:

Шаг 1: Вычисляем определитель

$$\det(A) = 2 \cdot 3 - 1 \cdot 1 = 6 - 1 = 5 \neq 0$$

Шаг 2: Находим алгебраические дополнения

$$A_{11} = (-1)^{1+1} \cdot 3 = 3, \quad A_{12} = (-1)^{1+2} \cdot 1 = -1$$
$$A_{21} = (-1)^{2+1} \cdot 1 = -1, \quad A_{22} = (-1)^{2+2} \cdot 2 = 2$$

Шаг 3: Составляем присоединённую матрицу

$$\text{adj}(A) = \begin{pmatrix} A_{11} & A_{21} \\ A_{12} & A_{22} \end{pmatrix} = \begin{pmatrix} 3 & -1 \\ -1 & 2 \end{pmatrix}$$

Шаг 4: Вычисляем обратную матрицу

$$A^{-1} = \frac{1}{5} \begin{pmatrix} 3 & -1 \\ -1 & 2 \end{pmatrix} = \begin{pmatrix} 0.6 & -0.2 \\ -0.2 & 0.4 \end{pmatrix}$$

Проверка:

$$AA^{-1} = \begin{pmatrix} 2 & 1 \\ 1 & 3 \end{pmatrix} \begin{pmatrix} 0.6 & -0.2 \\ -0.2 & 0.4 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

1.8.2 Метод Гаусса-Жордана

Определение 1.19. Метод Гаусса-Жордана

Метод основан на элементарных преобразованиях строк расширенной матрицы $(A|E)$ до вида $(E|A^{-1})$.

Найдём обратную матрицу для $A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$:

Шаг 1: Составляем расширенную матрицу

$$(A|E) = \left(\begin{array}{cc|cc} 1 & 2 & 1 & 0 \\ 3 & 4 & 0 & 1 \end{array}\right)$$

Шаг 2: Вычитаем из второй строки первую, умноженную на 3

$$\text{row}_2 \to \text{row}_2 - 3 \cdot \text{row}_1 : \quad \left(\begin{array}{cc|cc} 1 & 2 & 1 & 0 \\ 0 & -2 & -3 & 1 \end{array}\right)$$

Шаг 3: Делим вторую строку на -2

$$\text{row}_2 \to -\frac{1}{2} \cdot \text{row}_2 : \quad \left(\begin{array}{cc|cc} 1 & 2 & 1 & 0 \\ 0 & 1 & 1.5 & -0.5 \end{array}\right)$$

Шаг 4: Вычитаем из первой строки вторую, умноженную на 2

$$\text{row}_1 \to \text{row}_1 - 2 \cdot \text{row}_2 : \quad \left(\begin{array}{cc|cc} 1 & 0 & -2 & 1 \\ 0 & 1 & 1.5 & -0.5 \end{array}\right)$$

Результат:

$$A^{-1} = \begin{pmatrix} -2 & 1 \\ 1.5 & -0.5 \end{pmatrix}$$

Проверка:

$$AA^{-1} = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \begin{pmatrix} -2 & 1 \\ 1.5 & -0.5 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

Свойство 1.7. Свойства обратной матрицы

- $(A^{-1})^{-1} = A$
- $(AB)^{-1} = B^{-1}A^{-1}$
- $(A^T)^{-1} = (A^{-1})^T$
- $\det(A^{-1}) = \frac{1}{\det(A)}$
- $(\lambda A)^{-1} = \frac{1}{\lambda}A^{-1}$ для $\lambda \neq 0$

2. Системы линейных алгебраических уравнений

Определение 2.1. Система линейных уравнений

Система из n линейных уравнений с m неизвестными имеет вид:

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1m}x_m = b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2m}x_m = b_2 \\ \vdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nm}x_m = b_n \end{cases}$$

где $x_1, x_2, \ldots, x_m$ — неизвестные, a_{ij} — коэффициенты, b_i — свободные члены.

Определение 2.2. Матричная форма СЛАУ

Систему можно записать в матричном виде:

$$AX = B$$

где

$$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1m} \\ a_{21} & a_{22} & \cdots & a_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nm} \end{pmatrix}_{n\times m}, \quad X = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{pmatrix}_{m\times 1}, \quad B = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}_{n\times 1}$$

2.1. Метод обратной матрицы

Теорема 2.1. Решение через обратную матрицу

Если матрица A квадратная ($n = m$) и $\det(A) \neq 0$, то система имеет единственное решение:

$$X = A^{-1}B$$

Решим систему методом обратной матрицы:

$$\begin{cases} 2x + y = 5 \\ x + 3y = 10 \end{cases}$$

Матричная форма:

$$\begin{pmatrix} 2 & 1 \\ 1 & 3 \end{pmatrix}_{2\times 2} \begin{pmatrix} x \\ y \end{pmatrix}_{2\times 1} = \begin{pmatrix} 5 \\ 10 \end{pmatrix}_{2\times 1}$$

Находим обратную матрицу:

$$A^{-1} = \frac{1}{5}\begin{pmatrix} 3 & -1 \\ -1 & 2 \end{pmatrix} = \begin{pmatrix} 0.6 & -0.2 \\ -0.2 & 0.4 \end{pmatrix}_{2\times 2}$$

Решение:

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 0.6 & -0.2 \\ -0.2 & 0.4 \end{pmatrix}\begin{pmatrix} 5 \\ 10 \end{pmatrix} = \begin{pmatrix} 3-2 \\ -1+4 \end{pmatrix} = \begin{pmatrix} 1 \\ 3 \end{pmatrix}$$

Проверка: $2 \cdot 1 + 3 = 5$, $1 + 3 \cdot 3 = 10$

2.2. Метод Крамера

Определение 2.3. Формулы Крамера

Для системы n уравнений с n неизвестными ($n = m$), если $\Delta = \det(A) \neq 0$, то решение находится по формулам:

$$x_i = \frac{\Delta_i}{\Delta}, \quad i = 1, 2, \dots, n$$

где Δ — определитель основной матрицы A, а Δ_i — определитель матрицы, полученной заменой i-го столбца матрицы A на столбец свободных членов B.

Решим систему методом Крамера:

$$\begin{cases} x_1 + 2x_2 + x_3 = 9 \\ 2x_1 - x_2 + 3x_3 = 8 \\ 3x_1 + x_2 - 2x_3 = 1 \end{cases}$$

Основной определитель:

$$\Delta = \begin{vmatrix} 1 & 2 & 1 \\ 2 & -1 & 3 \\ 3 & 1 & -2 \end{vmatrix}_{3\times3} = 1 \cdot (-1) \cdot (-2) + 2 \cdot 3 \cdot 3 + 1 \cdot 2 \cdot 1 = -1 \cdot (-1) \cdot 3 - 2 \cdot 2 \cdot (-2) - 1 \cdot 3 \cdot 1 =$$
$$= 2 + 18 + 2 + 3 + 8 - 3 = 30$$

Определители для неизвестных:

$$\Delta_1 = \begin{vmatrix} 9 & 2 & 1 \\ 8 & -1 & 3 \\ 1 & 1 & -2 \end{vmatrix}_{3\times3} = 9 \cdot (-1) \cdot (-2) + 2 \cdot 3 \cdot 1 + 1 \cdot 8 \cdot 1 = -1 \cdot (-1) \cdot 1 - 2 \cdot 8 \cdot (-2) - 9 \cdot 3 \cdot 1 =$$
$$= 18 + 6 + 8 + 1 + 32 - 27 = 38$$

$$\Delta_2 = \begin{vmatrix} 1 & 9 & 1 \\ 2 & 8 & 3 \\ 3 & 1 & -2 \end{vmatrix}_{3\times3} = 1 \cdot 8 \cdot (-2) + 9 \cdot 3 \cdot 3 + 1 \cdot 2 \cdot 1 = -1 \cdot 8 \cdot 3 - 9 \cdot 2 \cdot (-2) - 1 \cdot 3 \cdot 1 =$$
$$= -16 + 81 + 2 - 24 + 36 - 3 = 76$$

$$\Delta_3 = \begin{vmatrix} 1 & 2 & 9 \\ 2 & -1 & 8 \\ 3 & 1 & 1 \end{vmatrix}_{3\times3} = 1 \cdot (-1) \cdot 1 + 2 \cdot 8 \cdot 3 + 9 \cdot 2 \cdot 1 = -9 \cdot (-1) \cdot 3 - 2 \cdot 2 \cdot 1 - 1 \cdot 8 \cdot 1 =$$
$$= -1 + 48 + 18 + 27 - 4 - 8 = 80$$

Решение:

$$x_1 = \frac{\Delta_1}{\Delta} = \frac{38}{30} = \frac{19}{15}, \quad x_2 = \frac{\Delta_2}{\Delta} = \frac{76}{30} = \frac{38}{15}, \quad x_3 = \frac{\Delta_3}{\Delta} = \frac{80}{30} = \frac{8}{3}$$

2.3. Метод Гаусса

Определение 2.4. Метод Гаусса

Метод последовательного исключения неизвестных, основанный на приведении расширенной матрицы системы к ступенчатому виду с помощью элементарных преобразований строк.

Теорема 2.2. Элементарные преобразования

Разрешённые преобразования, не меняющие множество решений системы:

- Перестановка двух уравнений
- Умножение уравнения на ненулевое число
- Прибавление к одному уравнению другого, умноженного на число

Решим систему методом Гаусса:

$$\begin{cases} 2x + 4y - z = 5 \\ x + 3y + 2z = 10 \\ 3x - y + z = 2 \end{cases}$$

Расширенная матрица:

$$\left(\begin{array}{ccc|c} 2 & 4 & -1 & 5 \\ 1 & 3 & 2 & 10 \\ 3 & -1 & 1 & 2 \end{array}\right)_{3\times 4}$$

Первый шаг: меняем местами первую и вторую строки

$$\left(\begin{array}{ccc|c} 1 & 3 & 2 & 10 \\ 2 & 4 & -1 & 5 \\ 3 & -1 & 1 & 2 \end{array}\right)$$

Второй шаг: row row - 2·row , row row - 3·row

$$\left(\begin{array}{ccc|c} 1 & 3 & 2 & 10 \\ 0 & -2 & -5 & -15 \\ 0 & -10 & -5 & -28 \end{array}\right)$$

Третий шаг: row - / ·row

$$\left(\begin{array}{ccc|c} 1 & 3 & 2 & 10 \\ 0 & 1 & 2.5 & 7.5 \\ 0 & -10 & -5 & -28 \end{array}\right)$$

Четвёртый шаг: row row + 10·row

$$\left(\begin{array}{ccc|c} 1 & 3 & 2 & 10 \\ 0 & 1 & 2.5 & 7.5 \\ 0 & 0 & 20 & 47 \end{array}\right)$$

Пятый шаг: row / ·row

$$\left(\begin{array}{ccc|c} 1 & 3 & 2 & 10 \\ 0 & 1 & 2.5 & 7.5 \\ 0 & 0 & 1 & 2.35 \end{array}\right)$$

Обратный ход:

$$z = 2.35$$

$$y + 2.5 \cdot 2.35 = 7.5 \Rightarrow y = 7.5 - 5.875 = 1.625$$

$$x + 3 \cdot 1.625 + 2 \cdot 2.35 = 10 \Rightarrow x = 10 - 4.875 - 4.7 = 0.425$$

Определение 2.5. Классификация СЛАУ

- **Определённая система**: единственное решение ($n = m$ и $\det(A) \neq 0$)
- **Неопределённая система**: бесконечно много решений
- **Несовместная система**: нет решений ($n < m$, количество уравнений меньше количества неизвестных)

3. Основы теории вероятностей

3.1. Основные понятия теории вероятностей

Определение 3.1. Вероятность

Вероятность — числовая характеристика степени возможности наступления некоторого события при определённых условиях. Обозначается $P(A)$ — вероятность события A.

Определение 3.2. Случайный эксперимент

Процесс, который может быть повторён многократно в одинаковых условиях, но результат которого непредсказуем. Примеры: бросок монеты, игральная кость, выбор карты из колоды.

Определение 3.3. Пространство элементарных исходов

Множество всех возможных результатов случайного эксперимента. Обозначается Ω.

Примеры:

- Бросок монеты: $\Omega = \{\text{орёл}, \text{решка}\}$
- Бросок игральной кости: $\Omega = \{1, 2, 3, 4, 5, 6\}$
- Два броска монеты: $\Omega = \{\text{OO}, \text{OP}, \text{PO}, \text{PP}\}$

Определение 3.4. Случайное событие

Любое подмножество пространства элементарных исходов. Событие происходит, если реализуется любой из входящих в него элементарных исходов.

Примеры:

- A: «выпадение чётного числа» при броске кости: $A = \{2, 4, 6\}$
- B: «выпадение решки» при броске монеты: $B = \{\text{решка}\}$

3.2. Комбинаторика в теории вероятностей

Определение 3.5. Факториал

Факториал натурального числа n — произведение всех натуральных чисел от 1 до n:

$$n! = 1 \cdot 2 \cdot 3 \cdot \dots \cdot (n-1) \cdot n$$

По определению: $0! = 1$

Примеры факториалов:

$$1! = 1, \quad 2! = 2, \quad 3! = 6, \quad 4! = 24, \quad 5! = 120, \quad 6! = 720$$

Определение 3.6. Перестановки

Число способов упорядочить n различных объектов:

$$P_n = n!$$

Пример: Сколькими способами можно расставить 4 различные книги на полке?

$$P_4 = 4! = 24 \text{ способа}$$

Определение 3.7. Перестановки с повторениями

Число различных перестановок n объектов, среди которых есть одинаковые:

$$P_n(n_1, n_2, \dots, n_k) = \frac{n!}{n_1! \cdot n_2! \cdot \dots \cdot n_k!}$$

где $n_1 + n_2 + \dots + n_k = n$

Пример: Сколько различных слов можно составить из букв слова "МАМА"?

$$P_4(2,2) = \frac{4!}{2! \cdot 2!} = \frac{24}{4} = 6 \text{ слов}$$

Пример: Сколько различных последовательностей можно составить из цифр 1, 1, 1, 2, 2?

$$P_5(3,2) = \frac{5!}{3! \cdot 2!} = \frac{120}{6 \cdot 2} = 10 \text{ последовательностей}$$

Определение 3.8. Размещения

Число способов выбрать и упорядочить k объектов из n различных:

$$A_n^k = \frac{n!}{(n-k)!} = n \cdot (n-1) \cdot \dots \cdot (n-k+1)$$

Пример: Сколькими способами можно выбрать 1-е, 2-е и 3-е места из 5 участников?

$$A_5^3 = 5 \cdot 4 \cdot 3 = 60 \text{ способов}$$

Пример: Сколько трёхзначных чисел можно составить из цифр 1, 2, 3, 4, 5 без повторений?

$$A_5^3 = 5 \cdot 4 \cdot 3 = 60 \text{ чисел}$$

Определение 3.9. Размещения с повторениями

Число способов выбрать и упорядочить k объектов из n с возможностью повторений:

$$\bar{A}_n^k = n^k$$

Пример: Сколько трёхзначных чисел можно составить из цифр 1, 2, 3, 4, 5 с повторениями?

$$\bar{A}_5^3 = 5^3 = 125 \text{ чисел}$$

Пример: Сколько различных последовательностей из 4 бросков монеты?

$$\bar{A}_2^4 = 2^4 = 16 \text{ последовательностей}$$

Определение 3.10. Сочетания

Число способов выбрать k объектов из n различных без учёта порядка:

$$C_n^k = \binom{n}{k} = \frac{n!}{k!(n-k)!} = \frac{A_n^k}{k!}$$

Пример: Сколькими способами можно выбрать 3 книги из 5?

$$C_5^3 = \frac{5!}{3! \cdot 2!} = \frac{120}{6 \cdot 2} = 10 \text{ способов}$$

Пример: Сколькими способами можно выбрать 2 человек из 4 для комитета?

$$C_4^2 = \frac{4!}{2! \cdot 2!} = \frac{24}{4} = 6 \text{ способов}$$

Определение 3.11. Сочетания с повторениями

Число способов выбрать k объектов из n типов с возможностью повторений:

$$\bar{C}_n^k = C_{n+k-1}^k = \frac{(n+k-1)!}{k!(n-1)!}$$

Пример: Сколькими способами можно выбрать 3 пирожных из 4 видов?

$$\bar{C}_4^3 = C_{4+3-1}^3 = C_6^3 = \frac{6!}{3! \cdot 3!} = \frac{720}{6 \cdot 6} = 20 \text{ способов}$$

Свойство 3.1. Свойства сочетаний

- $C_n^k = C_n^{n-k}$ (симметрия)
- $C_n^0 = C_n^n = 1$
- $C_n^1 = C_n^{n-1} = n$
- $C_n^k = C_{n-1}^{k-1} + C_{n-1}^k$ (рекуррентная формула)
- $\sum_{k=0}^{n} C_n^k = 2^n$

Пример применения свойства симметрии:

$$C_8^5 = C_8^3 = \frac{8 \cdot 7 \cdot 6}{3 \cdot 2 \cdot 1} = 56$$

Пример применения рекуррентной формулы:

$$C_5^3 = C_4^2 + C_4^3 = 6 + 4 = 10$$

3.3. Классическое определение вероятности

Определение 3.12. Формула классической вероятности

Если все элементарные исходы равновозможны, то вероятность события A равна:

$$P(A) = \frac{\text{число благоприятных исходов}}{\text{общее число исходов}} = \frac{|A|}{|\Omega|}$$

Пример: Вероятность выпадения чётного числа на игральной кости:

$$P(A) = \frac{3}{6} = \frac{1}{2}$$

Пример: Вероятность выпадения суммы 7 при броске двух костей:

$$P(\text{сумма} = 7) = \frac{6}{36} = \frac{1}{6}$$

Пример: Вероятность вытащить туза из колоды в 52 карты:

$$P(\text{туз}) = \frac{4}{52} = \frac{1}{13}$$

Пример с использованием комбинаторики: В группе из 10 человек нужно выбрать 3 для комитета. Какова вероятность, что будут выбраны конкретные 3 человека?

$$P = \frac{1}{C_{10}^3} = \frac{1}{120}$$

Пример: Какова вероятность получить 3 решки при 5 бросках монеты?

$$P = \frac{C_5^3}{2^5} = \frac{10}{32} = \frac{5}{16}$$

3.4. Операции над событиями

Определение 3.13. Объединение событий

Событие $A \cup B$ происходит, когда происходит хотя бы одно из событий A или B.

Определение 3.14. Пересечение событий

Событие $A \cap B$ происходит, когда происходят оба события A и B одновременно.

Определение 3.15. Противоположное событие

Событие $\bar{A}$ происходит тогда и только тогда, когда не происходит событие A.

Определение 3.16. Несовместные события

События A и B называются несовместными, если они не могут произойти одновременно: $A \cap B = \emptyset$.

Определение 3.17. Полная группа событий

События $A_1, A_2, \ldots, A_n$ образуют полную группу, если они попарно несовместны и их объединение даёт всё пространство исходов.

3.5. Теоремы сложения вероятностей

Теорема 3.1. Сложение вероятностей

Для любых двух событий A и B:

$$P(A \cup B) = P(A) + P(B) - P(A \cap B)$$

Теорема 3.2. Сложение для несовместных событий

Если события A и B несовместны, то:

$$P(A \cup B) = P(A) + P(B)$$

Теорема 3.3. Вероятность противоположного события

$$P(\bar{A}) = 1 - P(A)$$

Пример: Вероятность выпадения чётного числа или числа, большего 3, на игральной кости:

$$P(\text{чётное}) = \frac{3}{6}, \quad P(>3) = \frac{3}{6}, \quad P(\text{чётное и } >3) = \frac{1}{6}$$

$$P(\text{чётное или } >3) = \frac{3}{6} + \frac{3}{6} - \frac{1}{6} = \frac{5}{6}$$

Пример с несовместными событиями: Вероятность выпадения 2 или 5 на игральной кости:

$$P(2 \cup 5) = P(2) + P(5) = \frac{1}{6} + \frac{1}{6} = \frac{1}{3}$$

Пример с противоположным событием: Вероятность не выпадения шестёрки на игральной кости:

$$P(\bar{6}) = 1 - P(6) = 1 - \frac{1}{6} = \frac{5}{6}$$

3.6. Условная вероятность и независимость

Определение 3.18. Условная вероятность

Вероятность события A при условии, что событие B уже произошло:

$$P(A|B) = \frac{P(A \cap B)}{P(B)}, \quad \text{где } P(B) > 0$$

Определение 3.19. Независимые события

События A и B называются независимыми, если:

$$P(A \cap B) = P(A) \cdot P(B)$$

Эквивалентно: $P(A|B) = P(A)$ и $P(B|A) = P(B)$

Определение 3.20. Зависимые события

События называются зависимыми, если наступление одного влияет на вероятность наступления другого.

Пример условной вероятности: В колоде 52 карты. Вероятность вытащить туза при условии, что карта червовой масти:

$$P(\text{туз}|\text{черви}) = \frac{1}{13}$$

Пример независимых событий: Броски монеты - события независимы:

$$P(\text{орёл на 1-м броске} \cap \text{решка на 2-м броске}) = \frac{1}{2} \cdot \frac{1}{2} = \frac{1}{4}$$

Пример зависимых событий: В колоде 52 карты. Вероятность вытащить второго туза после того, как первый туз уже вытащен:

$$P(\text{второй туз}|\text{первый туз}) = \frac{3}{51}$$

3.7. Теорема умножения вероятностей

Теорема 3.4. Умножение вероятностей

Для любых двух событий:

$$P(A \cap B) = P(A) \cdot P(B|A) = P(B) \cdot P(A|B)$$

Теорема 3.5. Умножение для независимых событий

Если события A и B независимы, то:

$$P(A \cap B) = P(A) \cdot P(B)$$

Примеры:
Вероятность выпадения двух решек подряд (независимые события):

$$P(\text{РР}) = \frac{1}{2} \cdot \frac{1}{2} = \frac{1}{4}$$

Вероятность вытащить двух тузов подряд из колоды (зависимые события, без возвращения):

$$P(\text{два туза}) = \frac{4}{52} \cdot \frac{3}{51} = \frac{12}{2652} = \frac{1}{221}$$

Вероятность вытащить червового туза из колоды:

$$P(\text{туз и черви}) = \frac{1}{52}$$

Пример с тремя событиями: Вероятность вытащить трёх тузов подряд из колоды:

$$P(\text{три туза}) = \frac{4}{52} \cdot \frac{3}{51} \cdot \frac{2}{50} = \frac{24}{132600} = \frac{1}{5525}$$

Свойство 3.2. Свойства вероятности

- $0 \leq P(A) \leq 1$ для любого события A
- $P(\Omega) = 1$, $P(\emptyset) = 0$

- Если $A \subset B$, то $P(A) \leq P(B)$
- $P(A \cup B) \leq P(A) + P(B)$ (неравенство Буля)
- $P(A \cup B \cup C) = P(A) + P(B) + P(C) - P(A \cap B) - P(A \cap C) - P(B \cap C) + P(A \cap B \cap C)$

Пример применения свойства включения: Если A: "выпадение 2 B: "выпадение чётного числа то $A \subset B$ и:

$$P(A) = \frac{1}{6} \leq P(B) = \frac{1}{2}$$

Пример для трёх событий: Вероятность выпадения чётного числа ИЛИ числа больше 3 ИЛИ числа, делящегося на 3:

$$P(\text{чёт} \cup > 3 \cup \text{кратно } 3) = \frac{3}{6} + \frac{3}{6} + \frac{2}{6} - \frac{1}{6} - \frac{1}{6} - \frac{1}{6} + \frac{1}{6} = \frac{6}{6} = 1$$

4. Продвинутая теория вероятностей

4.1. Формула полной вероятности

Определение 4.1. Полная группа событий

События $H_1, H_2, \dots, H_n$ образуют полную группу, если они попарно несовместны и их объединение даёт всё пространство элементарных исходов:

$$H_i \cap H_j = \emptyset \quad \text{при } i \neq j, \quad \text{и} \quad H_1 \cup H_2 \cup \dots \cup H_n = \Omega$$

Определение 4.2. Формула полной вероятности

Если события $H_1, H_2, \dots, H_n$ образуют полную группу и $P(H_i) > 0$ для всех i, то для любого события A:

$$P(A) = \sum_{i=1}^{n} P(A \cap H_i) = \sum_{i=1}^{n} P(H_i) \cdot P(A|H_i)$$

Вывод формулы

Так как события $H_1, H_2, \dots, H_n$ образуют полную группу, то событие A можно представить как объединение попарно несовместных событий:

$$A = A \cap \Omega = A \cap (H_1 \cup H_2 \cup \dots \cup H_n) = (A \cap H_1) \cup (A \cap H_2) \cup \dots \cup (A \cap H_n)$$

Для попарно несовместных событий:

$$P(A) = P(A \cap H_1) + P(A \cap H_2) + \dots + P(A \cap H_n) = \sum_{i=1}^{n} P(A \cap H_i)$$

Применяя теорему умножения вероятностей $P(A \cap H_i) = P(H_i) \cdot P(A|H_i)$, получаем:

$$P(A) = \sum_{i=1}^{n} P(H_i) \cdot P(A|H_i)$$

Пример: На трёх заводах производят детали. Первый завод производит 30% всех деталей, второй — 45%, третий — 25%. Брак составляет 2% на первом заводе, 3% на втором и 4% на третьем. Найти вероятность того, что случайно выбранная деталь бракованная.

Введём гипотезы:

$$H_1 : \text{деталь с 1-го завода}, \quad P(H_1) = 0.30$$

$$H_2 : \text{деталь со 2-го завода}, \quad P(H_2) = 0.45$$

$$H_3 : \text{деталь с 3-го завода}, \quad P(H_3) = 0.25$$

Условные вероятности брака:

$$P(A|H_1) = 0.02, \quad P(A|H_2) = 0.03, \quad P(A|H_3) = 0.04$$

По формуле полной вероятности:

$$P(A) = 0.30 \cdot 0.02 + 0.45 \cdot 0.03 + 0.25 \cdot 0.04 = 0.006 + 0.0135 + 0.01 = 0.0295$$

4.2. Формула Байеса

Определение 4.3. Формула Байеса

Если события $H_1, H_2, \dots, H_n$ образуют полную группу и $P(H_i) > 0$ для всех i, то для любого события A с $P(A) > 0$:

$$P(H_i|A) = \frac{P(H_i) \cdot P(A|H_i)}{P(A)} = \frac{P(H_i) \cdot P(A|H_i)}{\sum_{j=1}^{n} P(H_j) \cdot P(A|H_j)}$$

Вывод формулы

По определению условной вероятности:

$$P(H_i|A) = \frac{P(H_i \cap A)}{P(A)}$$

Применяя теорему умножения в числителе и формулу полной вероятности в знаменателе:

$$P(H_i|A) = \frac{P(H_i) \cdot P(A|H_i)}{\sum_{j=1}^{n} P(H_j) \cdot P(A|H_j)}$$

Пример:

Продолжим пример с заводами. Найдём вероятность того, что бракованная деталь произведена на первом заводе.

Используем формулу Байеса:

$$P(H_1|A) = \frac{P(H_1) \cdot P(A|H_1)}{P(A)} = \frac{0.30 \cdot 0.02}{0.0295} = \frac{0.006}{0.0295} \approx 0.203$$

4.3. Схема испытаний Бернулли

Определение 4.4. Пространство элементарных исходов в схеме Бернулли

При проведении n независимых испытаний, каждое элементарное событие ω представляет собой вектор длины n:

$$\omega = (x_1, x_2, \dots, x_n), \quad \text{где } x_i \in \{0, 1\}$$

где $x_i = 1$, если в i-м испытании наступил успех, и $x_i = 0$ в противном случае.

Определение 4.5. Вероятность элементарного исхода

Для каждого элементарного исхода $\omega = (x_1, x_2, \dots, x_n)$ вероятность вычисляется как:

$$P(\omega) = p^k \cdot (1-p)^{n-k}$$

где k — количество единиц в векторе ω (число успехов), p — вероятность успеха в одном испытании.

Определение 4.6. Схема испытаний Бернулли

Последовательность n независимых испытаний, в каждом из которых событие A может наступить с одной и той же вероятностью $p = P(A)$, а не наступить с вероятностью $q = 1 - p$.

Определение 4.7. Формула Бернулли

Вероятность того, что в n испытаниях Бернулли событие A наступит ровно k раз:

$$P_n(k) = C_n^k \cdot p^k \cdot q^{n-k} = \frac{n!}{k!(n-k)!} \cdot p^k \cdot (1-p)^{n-k}$$

Вывод формулы

Событие A = «ровно k успехов в n испытаниях» состоит из всех элементарных исходов ω, содержащих ровно k единиц. Число таких исходов равно C_n^k. Каждый такой исход имеет вероятность $p^k \cdot (1-p)^{n-k}$. Поэтому:

$$P(A) = C_n^k \cdot p^k \cdot (1-p)^{n-k}$$

Пример:

Монету подбрасывают 5 раз. Найти вероятность выпадения ровно 3 орлов.

$$n = 5, \quad k = 3, \quad p = 0.5, \quad q = 0.5$$

Пространство элементарных исходов состоит из векторов длины 5 из 0 и 1. Событие «ровно 3 орла» состоит из $C_5^3 = 10$ элементарных исходов, каждый с вероятностью $(0.5)^3 \cdot (0.5)^2 = 0.03125$:

$$P_5(3) = C_5^3 \cdot (0.5)^3 \cdot (0.5)^2 = 10 \cdot 0.125 \cdot 0.25 = 10 \cdot 0.03125 = 0.3125$$

Пример:

Стрелок попадает в цель с вероятностью 0.8. Производится 4 выстрела. Найти вероятность ровно 3 попаданий.

$$n = 4, \quad k = 3, \quad p = 0.8, \quad q = 0.2$$

$$P_4(3) = C_4^3 \cdot (0.8)^3 \cdot (0.2)^1 = 4 \cdot 0.512 \cdot 0.2 = 4 \cdot 0.1024 = 0.4096$$

5. Основы машинного обучения. Линейные модели

5.1. Задачи машинного обучения

Машинное обучение — это область, где моделируется зависимость между входными признаками и целевой переменной. Ключевые задачи:

Определение 5.1. Классификация

Цель: разделить объекты на заранее известные классы (например, определить болезнь по симптомам).

Определение 5.2. Регрессия

Цель: спрогнозировать числовое значение, исходя из набора признаков (например, предсказать цену квартиры).

Определение 5.3. Кластеризация

Цель: разбить объекты на группы по схожести, не используя заранее известных меток (например, анализ поведения клиентов банка).

5.2. Обучающая выборка

При построении модели машинного обучения крайне важно правильно организовать работу с данными. Все имеющиеся данные делятся на несколько частей, каждая из которых выполняет свою роль в процессе создания и оценки модели.

Определение 5.4. Обучающая выборка (Training Set)

Обучающая выборка — это часть исходных данных, которая используется непосредственно для подбора параметров модели. На этих данных алгоритм «учится», находя зависимости между входными признаками и целевой переменной.

Роль обучающей выборки

Обучающая выборка — это фундамент, на котором строится вся модель. Именно на этих данных происходит:

- Подбор параметров (весов): В линейной регрессии это коэффициенты $w_0, w_1, \ldots, w_n$, которые минимизируют функцию потерь на обучающих данных
- Выявление закономерностей: Модель анализирует связи между признаками и целевой переменной
- Обучение правил принятия решений: В классификации — определение разделяющей границы между классами

Размер обучающей выборки

Количество данных в обучающей выборке напрямую влияет на качество модели:

- Малая выборка (десятки примеров): модель может не уловить истинную зависимость, высок риск переобучения
- Средняя выборка (сотни-тысячи примеров): достаточно для большинства простых задач с небольшим числом признаков

- Большая выборка (десятки-сотни тысяч примеров): позволяет обучать сложные модели с большим числом параметров

Эмпирическое правило: для линейной модели желательно иметь как минимум в 5-10 раз больше примеров, чем признаков, чтобы избежать переобучения.

Определение 5.5. Тестовая выборка (Test Set)

Тестовая выборка — это часть данных, которая **не участвует в обучении** и используется исключительно для финальной оценки качества модели на ранее не виденных данных. Эта выборка имитирует реальные условия применения модели.

Определение 5.6. Валидационная выборка (Validation Set)

Валидационная выборка — это дополнительная часть данных, выделяемая из обучающей выборки для настройки гиперпараметров модели (например, степени полинома в полиномиальной регрессии) и промежуточной оценки качества в процессе обучения.

Выбор пропорции зависит от общего объёма данных: чем меньше данных, тем больше нужно выделить на обучение, но при этом сохранить достаточный объём для тестирования.

Замечание 2. Золотое правило машинного обучения

Никогда не используйте тестовую выборку для принятия решений об архитектуре модели или подборе параметров!
Тестовая выборка должна использоваться только один раз — для финальной оценки качества уже обученной модели. Если многократно оценивать разные варианты модели на тестовой выборке и на основе этого выбирать лучший вариант, то тестовая выборка фактически становится частью процесса обучения, и оценка качества перестаёт быть объективной.

Проблема при малом объёме данных:

При ограниченном количестве данных фиксированное разделение на 80/20 может быть неэффективным. Если у нас всего 50 примеров, то на обучение приходится только 40 примеров — этого может быть недостаточно для хорошего обучения. Одновременно тестовая выборка из 10 примеров может быть статистически неустойчивой: оценка качества будет сильно зависеть от того, какие конкретные примеры попали в тест. Кроме того, при экспериментах с гиперпараметрами (выбор степени полинома, количество признаков и т.д.) нам нужна валидационная выборка, но её выделение ещё больше сокращает размер обучающего набора.

Варианты разделения выборки и оценка переобучения

При построении модели нужно не только правильно разделить данные, но и выбрать метод, который позволит достоверно оценить, переобучилась ли модель. Существует несколько стратегий разделения данных, каждая с собственными преимуществами и недостатками.

Определение 5.7. Hold-Out разделение

Hold-Out (удержание) — это самый простой и быстрый метод разделения данных. Исходная выборка делится на две части: обучающую и тестовую. После обучения модели на первой части она оценивается на второй части. Это одноразовое разделение, и оценка качества получается в результате одного проведённого теста.

Типичные пропорции Hold-Out разделения:

- **80/20:** 80% данных — обучение, 20% — тестирование (самая популярная схема для среднего объёма данных, 100-10000 примеров)
- **70/30:** 70% — обучение, 30% — тестирование (используется при небольших данных, когда нужен больший тест для стабильности оценки)
- **90/10:** 90% — обучение, 10% — тестирование (используется при очень больших данных, когда даже 10% составляют тысячи примеров)
- **60/40:** 60% — обучение, 40% — тестирование (редко, если данных очень мало и нужна максимально надёжная оценка)

Hold-Out разделение привлекает простотой: достаточно один раз разбить данные и обучить модель. Однако оценка качества существенно зависит от того, какие примеры случайно попали в тест. Если в тестовую выборку попадут необычные примеры, оценка будет завышена; если попадут типичные — может быть занижена. Кроме того, если модель обучалась на 80% данных, а её работоспособность оценивалась на оставшихся 20%, это не отражает полностью её потенциал на большем объёме обучающих примеров.

Трёхразовое разделение для подбора гиперпараметров

Когда нужно выбирать гиперпараметры модели (например, степень полинома в полиномиальной регрессии), используют трёхразовое разделение:

- **70/15/15:** 70% — обучение, 15% — валидация, 15% — тестирование

Здесь валидационная выборка используется для подбора параметров: перепробуем разные варианты модели, оценим каждый на валидационной выборке и выберем лучший. Только после этого финальную модель оценивают на совершенно независимой тестовой выборке, которая не участвовала в выборе параметров.

Проблема Hold-Out с малыми данными

Главная проблема Hold-Out метода проявляется при малом объёме данных. Если у нас всего 50 примеров, то обучение происходит на 40 примерах, а тестирование — на 10. Десять примеров — очень маленькая выборка для надёжной оценки. Кроме того, случайное распределение примеров может создать смещение: в обучающую выборку могут случайно попасть в основном «лёгкие» примеры, а в тестовую — «сложные», что завысит оценку ошибки.

Чтобы получить более стабильную оценку, нужно использовать методы, которые не выбрасывают ни один пример и не полагаются на одно случайное разбиение.

Определение 5.8. Скользящий контроль (leave-one-out)

Скользящий контроль — это метод, при котором из исходной выборки исключается по очереди ровно один вектор (пример). Модель обучается на оставшихся $m - 1$ примерах, а затем проверяется на этом одном исключённом примере. Процесс повторяется m раз (по количеству объектов выборки), каждый раз оставляя другой пример для тестирования. Получив m оценок качества $a_1(x), a_2(x), \dots, a_m(x)$ (по одной для каждого примера), их усредняют: $\alpha(x) = \frac{1}{m} \sum_{i=1}^{m} a_i(x)$.

Скользящий контроль использует все данные максимально полно: каждый пример участвует в обучении $m - 1$ раз и в тестировании ровно один раз. Это даёт несмещённую оценку качества модели. Однако требует обучения модели m раз, что при больших m (например, $m = 10000$) становится вычислительно непозволительно дорого. К тому же оценка получается очень нестабильной.

Определение 5.9. Кросс-валидация (k-fold, cross-validation)

Кросс-валидация — компромиссный метод между hold-out и скользящим контролем. Исходная выборка разбивается на k примерно равных частей (складок, folds). Затем выполняются k итераций: на каждой итерации одна складка используется как тестовая, остальные $k-1$ складок — как обучающие. После каждой итерации вычисляется ошибка на тестовой складке. Итоговая оценка качества получается усреднением ошибок по всем k итерациям: $\alpha(x) = \frac{1}{k}\sum_{i=1}^{k} \text{Error}_i$.

Как кросс-валидация работает на практике:

Рассмотрим выборку из 30 примеров с использованием 5-fold кросс-валидации. Разбиваем данные на 5 равных групп по 6 примеров каждая:

1. Группа 1 — тест (6 примеров), Группы 2-5 — обучение (24 примера). Обучаем модель, считаем ошибку на группе 1.
2. Группа 2 — тест, Группы 1,3,4,5 — обучение. Считаем ошибку на группе 2.
3. Группа 3 — тест, остальные — обучение. Считаем ошибку на группе 3.
4. Группа 4 — тест, остальные — обучение. Считаем ошибку на группе 4.
5. Группа 5 — тест, остальные — обучение. Считаем ошибку на группе 5.

Получили 5 оценок ошибок: Error_1, Error_2, Error_3, Error_4, Error_5. Усредняя их, получаем финальную оценку: $\alpha(x) = \frac{1}{5}(Error_1 + Error_2 + ... + Error_5)$.

Преимущества кросс-валидации:

- Полное использование данных: Каждый пример используется для обучения в $k-1$ итерациях (при $k=5$ это 4 раза) и для тестирования в 1 итерации. Ничего не выбрасывается.
- Более стабильная оценка: Вместо одного случайного разбиения (как в Hold-Out) получаем k разных разбиений и усредняем результаты. Это снижает влияние случайности.
- Статистическая информация: Получаем k независимых оценок, что позволяет вычислить стандартное отклонение оценки и построить доверительный интервал.
- Меньше вычислений, чем скользящий контроль: Обучаем модель k раз (например, 5 раз), а не m раз (например, 1000 раз).

5.3. Функции потерь

В машинном обучении функция потерь $L(w)$ — это ключевой элемент, который количественно оценивает, насколько выдаваемые моделью ответы отличаются от целевых значений. Здесь w — совокупность внутренних параметров (весов) модели. Функция потерь принимает выходы модели и реальные метки, выдаёт число — меру ошибки, и тем самым позволяет управлять процессом обучения.

Смысл функции потерь состоит в том, чтобы задать численно, насколько плохо выбраны параметры модели w на данном наборе примеров. Если значения функции потерь высоки, модель работает плохо, и требуется корректировка параметров. Задача обучения — подобрать такие w, чтобы функция потерь достигала минимального значения.

Для одного объекта из обучающей выборки функция потерь может быть записана как:

$$L(w; x_i, y_i) = \ell(y_i, f_w(x_i))$$

где x_i — признаки объекта, y_i — целевая переменная, $f_w(x_i)$ — предсказание модели (с параметрами w), а ℓ — конкретная формула функции потерь.

На практике интересует не только ошибка на одном примере, а среднее качество на всех обучающих данных. В этом случае вводится понятие среднего эмпирического риска $Q(w)$:

Определение 5.10. Средний эмпирический риск

$$Q(w) = \frac{1}{n}\sum_{i=1}^{n} L(w; x_i, y_i)$$

Средний эмпирический риск показывает типичную ошибку, совершаемую моделью на обучающей выборке при заданных параметрах. Минимизация $Q(w)$ — это базовый принцип настройки любых моделей машинного обучения:

$$w^* = \arg\min_w Q(w)$$

5.4. Часто используемые функции потерь для задачи регрессии

Средний эмпирический риск — это общий принцип. На практике его конкретная форма зависит от того, какую функцию потерь ℓ мы выбираем. Рассмотрим основные варианты для задач регрессии.

Определение 5.11. Среднеквадратичная ошибка (MSE)

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

Среднеквадратичная ошибка — классический пример функции потерь для задач регрессии. Она вычисляет квадрат разности между предсказанием $\hat{y}_i$ и истинным значением y_i, затем усредняет по всей выборке. Метод удобен для анализа и оптимизации, а также усиливает влияние крупных ошибок за счёт возведения в квадрат. MSE подходит, когда крупные отклонения крайне нежелательны.

Минимизация MSE эквивалентна предположению о нормальном распределении ошибок модели, что связывает метод с принципом максимального правдоподобия в статистике. Это придаёт MSE теоретическое обоснование и объясняет её популярность.

Определение 5.12. Средняя абсолютная ошибка (MAE)

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|$$

Средняя абсолютная ошибка — альтернатива MSE для оценки качества моделей регрессии. Она измеряет модуль разности между реальным и предсказанным значением и усредняет по выборке. MAE менее чувствительна к сильным выбросам, поэтому хорошо работает с шумными данными. Её преимущества — интерпретируемость (результат в исходных единицах измерения), устойчивость к выбросам и простая связь с реальными ошибками.

Сравнение MSE и MAE:

MSE выбирают, когда необходимо усиленно штрафовать крупные ошибки, данные без существенных выбросов, нужен теоретический анализ и аналитическое решение (например, для линейной регрессии), требуется гладкая дифференцируемая функция для оптимизации.

MAE выбирают, когда важно получить типичную ошибку без преувеличения крупных промахов, данные содержат естественные выбросы, задача не суперчувствительна к редким крупным ошибкам, интерпретируемость важнее дифференцируемости, нужна оценка в исходных единицах.

5.5. Линейная регрессия

Определение 5.13. Линейная модель

Модель, которая описывает зависимость между наблюдениями и регрессорами с помощью линейной комбинации (веса w_i):

$$y = w_0 + w_1 x_1 + \cdots + w_n x_n + \varepsilon$$

где $w_0, w_1, \ldots, w_n$ — параметры модели (веса), $x_1, \ldots, x_n$ — признаки объекта, ε — случайная ошибка.

Линейная регрессия — один из старейших и наиболее изученных методов машинного обучения. Несмотря на свою простоту, она находит широкое применение благодаря интерпретируемос результатов, вычислительной эффективности и наличию аналитического решения. Линейная модель предполагает, что целевая переменная y является линейной комбинацией входных признаков с добавлением случайного шума.

Геометрическая интерпретация:

В многомерном случае линейная зависимость представляет собой гиперплоскость в пространстве признаков. Для случая одного признака ($n = 1$) это прямая линия на плоскости, для двух признаков ($n = 2$) — плоскость в трёхмерном пространстве, для большего числа признаков — гиперплоскость в многомерном пространстве.

Геометрическая задача линейной регрессии состоит в нахождении такой гиперплоскости, чтобы расстояния от всех точек данных до неё (ошибки предсказания) были минимальны в смысле выбранной функции потерь. Классический метод наименьших квадратов минимизирует сумму квадратов этих расстояний.

Пример 5.1. Аппроксимация данных линейной моделью

Пусть даны пары (x_i, y_i), например, площадь квартиры (в м^2) и её цена (в млн руб.). Если зависимость близка к линейной, то модель вида $y = w_0 + w_1 x$ хорошо опишет данные. Коэффициент w_1 показывает, на сколько в среднем увеличивается цена при увеличении площади на 1 м^2, а w_0 — базовую цену.

Если же зависимость нелинейная (например, квадратичная), то простая линейная модель даст большую ошибку. В таком случае помогает расширение признакового пространства — добавление новых признаков, таких как x^2, x^3 и т.д.

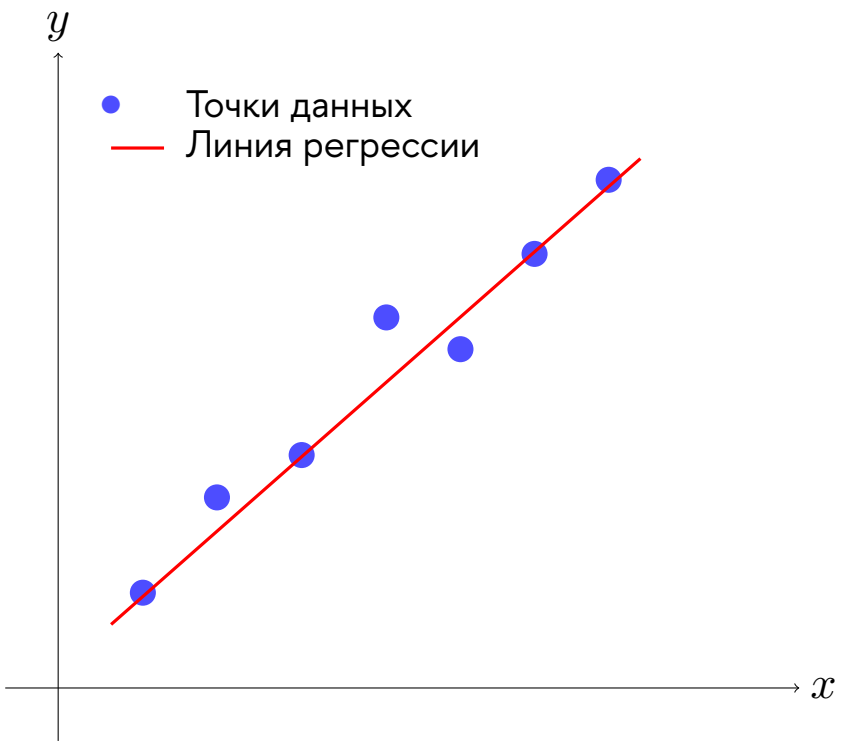


Рис. 1: Линейная регрессия: прямая минимизирует расстояния до точек

Матричная форма записи:

Для удобства работы с множеством объектов линейную регрессию записывают в матричной форме. Пусть имеется m объектов и n признаков:

$$X = \begin{pmatrix} 1 & x_{11} & x_{12} & \cdots & x_{1n} \\ 1 & x_{21} & x_{22} & \cdots & x_{2n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{m1} & x_{m2} & \cdots & x_{mn} \end{pmatrix}, \quad y = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{pmatrix}, \quad w = \begin{pmatrix} w_0 \\ w_1 \\ \vdots \\ w_n \end{pmatrix}$$

Здесь X — матрица признаков размера $m \times (n+1)$ (первый столбец из единиц соответствует свободному члену w_0), y — вектор истинных значений, w — вектор параметров модели.

Модель в матричном виде:

$$y = Xw + \varepsilon$$

Применение линейной регрессии:

Линейная регрессия используется для:

- Прогнозирования: предсказание значений целевой переменной для новых объектов (цены недвижимости, спроса на товар, температуры).
- Анализа зависимостей: оценка влияния каждого признака на целевую переменную через коэффициенты w_i. Положительный коэффициент означает прямую зависимость, отрицательный — обратную.
- Аппроксимации данных: построение линии тренда, описывающей общую закономерность в данных.
- Базовой модели (baseline): линейная регрессия часто служит отправной точкой для сравнения с более сложными моделями.

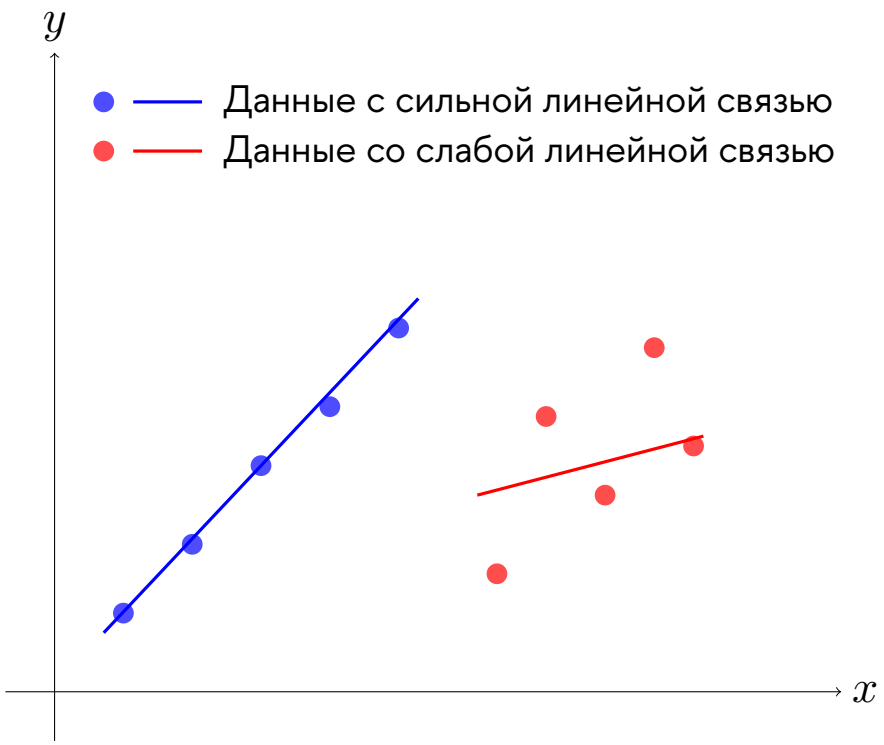


Рис. 2: Различные случаи линейной аппроксимации

Замечание 3. Ограничения линейной регрессии

Линейная регрессия предполагает линейную зависимость между признаками и целевой переменной. Если реальная зависимость существенно нелинейная, простая линейная модель будет давать большие ошибки. В таких случаях применяют:

- Полиномиальную регрессию (добавление степеней признаков)
- Нелинейные преобразования признаков (логарифм, экспонента и др.)
- Более сложные модели (деревья решений, нейронные сети)

Определение 5.14. Преимущества и недостатки линейной регрессии

Преимущества:

- Простота реализации и интерпретации
- Наличие аналитического решения (быстрое обучение)
- Низкая вычислительная сложность
- Хорошо работает при линейных или близких к линейным зависимостях
- Устойчивость к переобучению при небольшом числе признаков

Недостатки:

- Неспособность моделировать сложные нелинейные зависимости без преобразования признаков
- Чувствительность к выбросам (особенно при использовании MSE)

- Предположение о независимости признаков может нарушаться (мультиколлинеарность)

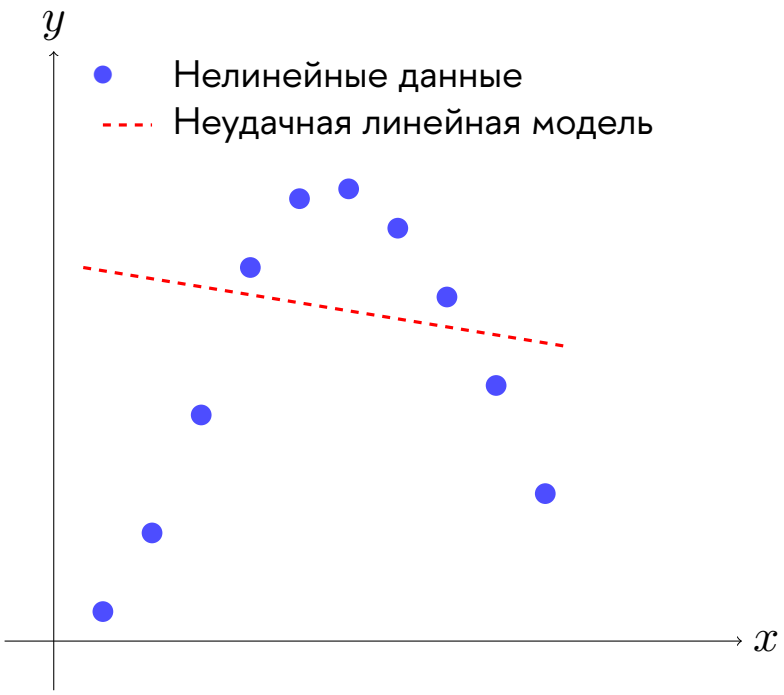


Рис. 3: Неудача линейной регрессии на нелинейных данных

5.6. Расширение признакового пространства и полиномиальная регрессия

Одна из ключевых идей машинного обучения заключается в том, что даже простые линейные модели могут описывать сложные нелинейные зависимости, если правильно преобразовать исходные признаки. Этот процесс называется **расширением признакового пространства**.

Определение 5.15. Полиномиальная регрессия

Полиномиальная регрессия — это метод, при котором к исходным признакам добавляются их степени $(x^2, x^3, \dots, x^d)$, что позволяет линейной модели аппроксимировать нелинейные зависимости:

$$y = w_0 + w_1 x + w_2 x^2 + \cdots + w_d x^d$$

где d — степень полинома, $w_0, w_1, \dots, w_d$ — параметры модели.

Важно понимать, что несмотря на название «полиномиальная регрессия», модель остаётся **линейной относительно параметров** w_i. Нелинейность присутствует только в зависимости от входной переменной x. Это позволяет использовать те же методы оптимизации, что и для обычной линейной регрессии, включая аналитическое решение через метод наименьших квадратов.

Геометрическая интерпретация:

При расширении признакового пространства мы переходим из исходного пространства признаков в пространство большей размерности. Например, если у нас был один признак x, то после добавления x^2 и x^3 мы получаем трёхмерное пространство признаков (x, x^2, x^3). В

этом новом пространстве данные могут стать линейно разделимыми или лучше аппроксимироваться гиперплоскостью.

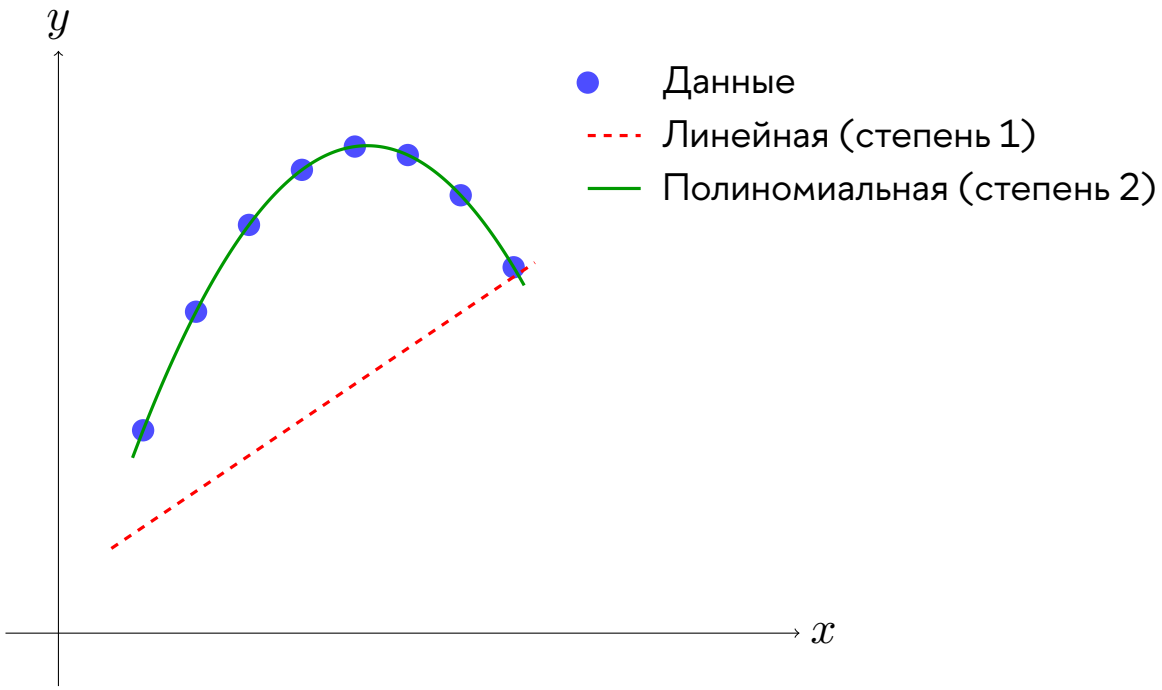


Рис. 4: Сравнение линейной и полиномиальной аппроксимации

Пример 5.2. Квадратичная аппроксимация

Предположим, что истинная зависимость между переменными имеет вид $y = 2 + 3x - 0.5x^2 + \varepsilon$, где ε — случайный шум. Линейная модель $y = w_0 + w_1 x$ не сможет точно описать эту зависимость, так как она не учитывает квадратичный член.
Если же построить полиномиальную регрессию степени 2, добавив признак x^2, модель примет вид $y = w_0 + w_1 x + w_2 x^2$. Теперь модель способна уловить квадратичную зависимость, и параметры w_0, w_1, w_2 будут близки к истинным коэффициентам $2, 3, -0.5$ соответственно.

Множественные признаки:

Расширение признакового пространства работает и для случая нескольких исходных признаков. Например, при двух признаках x_1 и x_2 полиномиальные признаки степени 2 включают: $x_1, x_2, x_1^2, x_1 x_2, x_2^2$. Это позволяет учитывать не только степени отдельных признаков, но и их взаимодействия (произведения).

Замечание 4. Важно: линейная независимость признаков

При расширении признакового пространства необходимо следить за тем, чтобы признаки оставались линейно независимыми. Дублирование информации (например, добавление признака $2x$ при наличии x) не увеличит информативность модели, но может привести к численным проблемам при вычислении обратной матрицы в методе наименьших квадратов.
Также стоит помнить о явлении **мультиколлинеарности** — высокой корреляции между признаками, которая может ухудшить устойчивость оценок параметров.

5.7. Аналитическое решение методом наименьших квадратов

Метод наименьших квадратов (МНК) — классический подход к нахождению параметров линейной регрессии, основанный на минимизации суммы квадратов отклонений предсказанных значений от истинных.

Определение 5.16. Метод наименьших квадратов

Параметры w выбираются так, чтобы сумма квадратов отклонений между реальными y_i и предсказанными $\hat{y}_i$ значениями была минимальной:

$$L(w) = \sum_{i=1}^{m}(y_i - \hat{y}_i)^2 = \sum_{i=1}^{m}(y_i - w^T x^{(i)})^2$$

В матричной форме (где X — матрица признаков размера $m \times (n+1)$, y — вектор истинных значений):

$$L(w) = \|y - Xw\|^2 = (y - Xw)^T(y - Xw)$$

Вывод аналитического решения:

Раскроем квадрат нормы:

$$L(w) = (y - Xw)^T(y - Xw) = y^T y - 2y^T Xw + w^T X^T Xw$$

Для нахождения минимума функции $L(w)$ необходимо найти точку, в которой производная (градиент) функции по вектору параметров w обращается в ноль. Вычислим производную по w:

$$\frac{\partial L(w)}{\partial w} = -2X^T y + 2X^T Xw$$

Приравнивая к нулю, получаем систему линейных уравнений, называемую **нормальным уравнением**:

$$X^T Xw = X^T y$$

Если матрица $X^T X$ невырождена (обратима), то решение имеет вид:

$$w = (X^T X)^{-1} X^T y$$

Это и есть аналитическое решение задачи линейной регрессии.

Геометрический смысл:

С геометрической точки зрения, метод наименьших квадратов находит ортогональную проекцию вектора y на подпространство, натянутое на столбцы матрицы X. Вектор остатков $e = y - Xw$ ортогонален этому подпространству, что и выражается условием $X^T e = 0$, откуда следует нормальное уравнение.

Условия применимости:

Аналитическое решение существует при выполнении следующих условий:

- Матрица $X^T X$ должна быть невырожденной (определитель $\det(X^T X) \neq 0$)
- Количество наблюдений m должно быть не меньше числа признаков $n+1$: $m \geq n+1$
- Признаки должны быть линейно независимыми

Если эти условия не выполняются, используют численные методы оптимизации (градиентный спуск) или регуляризацию.

Пример 5.3. Пример расчёта коэффициентов для простой линейной регрессии

Рассмотрим простейший случай с одним признаком. Пусть даны точки: $(x_1, y_1) = (1, 2)$, $(x_2, y_2) = (2, 4)$, $(x_3, y_3) = (3, 6)$.
Составим матрицу признаков (добавляя столбец из единиц для свободного члена):

$$X = \begin{pmatrix} 1 & 1 \\ 1 & 2 \\ 1 & 3 \end{pmatrix}, \quad y = \begin{pmatrix} 2 \\ 4 \\ 6 \end{pmatrix}$$

Вычислим $X^T X$ и $X^T y$:

$$X^T X = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 2 & 3 \end{pmatrix} \begin{pmatrix} 1 & 1 \\ 1 & 2 \\ 1 & 3 \end{pmatrix} = \begin{pmatrix} 3 & 6 \\ 6 & 14 \end{pmatrix}$$

$$X^T y = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 2 & 3 \end{pmatrix} \begin{pmatrix} 2 \\ 4 \\ 6 \end{pmatrix} = \begin{pmatrix} 12 \\ 28 \end{pmatrix}$$

Найдём обратную матрицу:

$$(X^T X)^{-1} = \frac{1}{3 \cdot 14 - 6 \cdot 6} \begin{pmatrix} 14 & -6 \\ -6 & 3 \end{pmatrix} = \frac{1}{6} \begin{pmatrix} 14 & -6 \\ -6 & 3 \end{pmatrix}$$

Наконец, вычислим параметры:

$$w = (X^T X)^{-1} X^T y = \frac{1}{6} \begin{pmatrix} 14 & -6 \\ -6 & 3 \end{pmatrix} \begin{pmatrix} 12 \\ 28 \end{pmatrix} = \frac{1}{6} \begin{pmatrix} 0 \\ 12 \end{pmatrix} = \begin{pmatrix} 0 \\ 2 \end{pmatrix}$$

Получили $w_0 = 0$, $w_1 = 2$, откуда уравнение регрессии: $y = 2x$. Прямая идеально проходит через все точки, что естественно, так как исходные данные лежат на прямой.

5.8. Переобучение и недообучение

При построении моделей машинного обучения важно найти баланс между простотой модели и её способностью описывать сложные зависимости в данных. Два противоположных явления — переобучение и недообучение — демонстрируют, что происходит, когда этот баланс нарушается.

Определение 5.17. Переобучение (Overfitting)

Переобучение возникает, когда модель слишком хорошо «подстраивается» под обучающие данные, запоминая не только истинную закономерность, но и случайный шум. Такая модель показывает отличные результаты на обучающей выборке, но плохо обобщается на новые данные.
Характерные признаки переобучения:

- Очень низкая ошибка на обучающей выборке
- Значительно более высокая ошибка на тестовой выборке
- Модель имеет слишком много параметров относительно объёма данных

Определение 5.18. Недообучение (Underfitting)

Недообучение происходит, когда модель слишком проста и не способна уловить истинную зависимость в данных. Такая модель показывает плохие результаты как на обучающей, так и на тестовой выборке.
Характерные признаки недообучения:

- Высокая ошибка на обучающей выборке
- Высокая ошибка на тестовой выборке
- Модель не улавливает очевидные закономерности в данных

Связь со степенью полинома:
В контексте полиномиальной регрессии степень полинома d является ключевым параметром, определяющим сложность модели:

- **Малые степени** ($d = 1, 2$): модель может быть слишком простой $\Rightarrow$ недообучение
- **Оптимальная степень**: модель улавливает основную закономерность без запоминания шума
- **Высокие степени** ($d \geq 5, 10, 15$): модель становится слишком гибкой $\Rightarrow$ переобучение

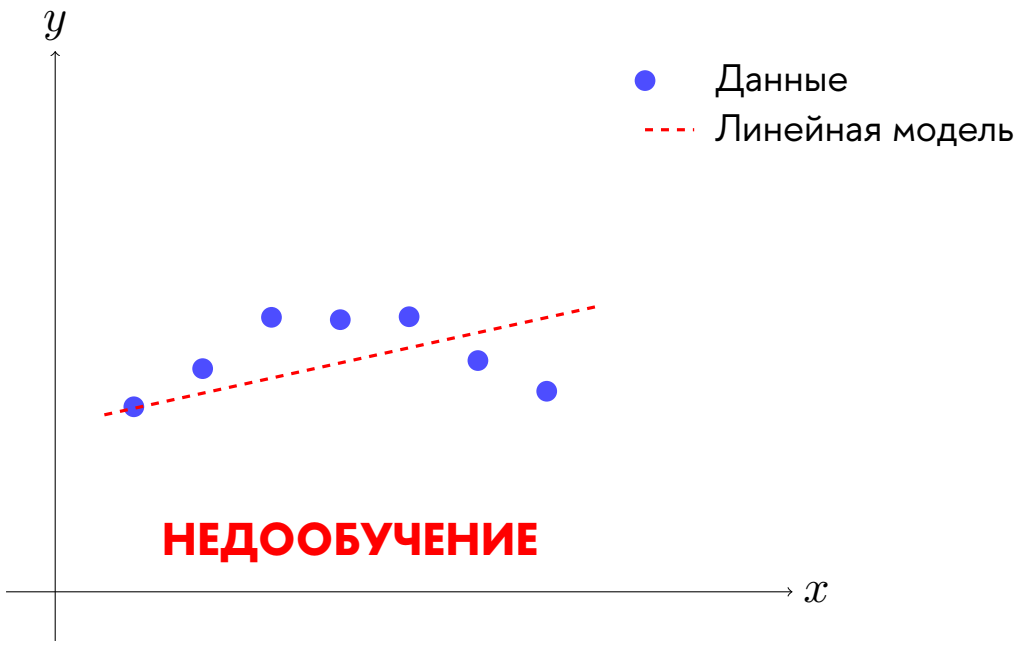


Рис. 5: Недообучение: линейная модель (степень 1) не улавливает нелинейную зависимость

Пример 5.4. Графическое сравнение моделей разной сложности

Рассмотрим набор данных с нелинейной зависимостью и небольшим шумом:

- **Степень 1 (линейная):** Модель представляет собой прямую, которая не может описать криволинейную структуру данных. Ошибка велика и на обучающей, и на тестовой выборке — *недообучение*.
- **Степень 2:** Модель хорошо аппроксимирует основной тренд, не подстраиваясь под шум. Ошибки на обучающей и тестовой выборках близки — *оптимальная*

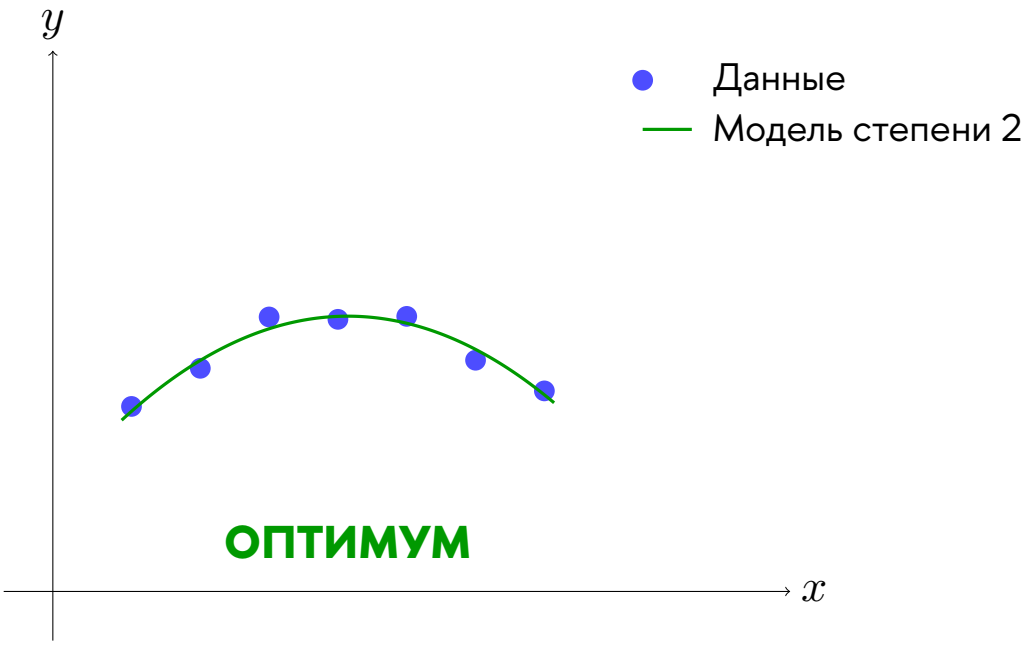


Рис. 6: Оптимальная модель (степень 2): улавливает основную закономерность, небольшие ошибки

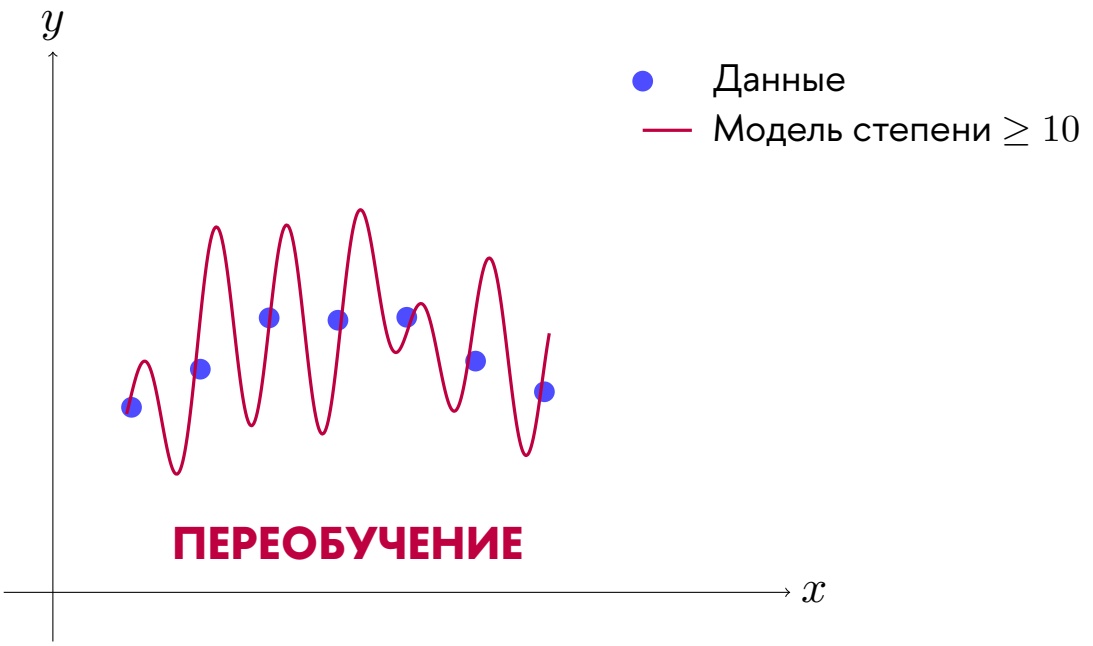


Рис. 7: Переобучение (степень ≥ 10): модель запоминает шум, делает излишние колебания между точками

сложность.

- **Степень $\geq$ 10:** Модель проходит очень близко к каждой точке обучающей выборки, делая резкие изгибы. На тестовой выборке такая модель даёт большие ошибки — *переобучение.*

Цель состоит в нахождении баланса: модель должна быть достаточно сложной, чтобы уловить закономерность (низкое смещение), но не настолько сложной, чтобы запоминать шум (низкая дисперсия).

Основные подходы к предотвращению переобучения:

- Увеличение объёма обучающих данных
- Регуляризация (добавление штрафа за сложность модели)
- Кросс-валидация для выбора оптимальной степени полинома
- Уменьшение числа признаков или их отбор
- Ранняя остановка обучения (для итеративных методов)

5.9. Линейная модель для бинарной классификации

До этого момента мы рассматривали линейную регрессию — задачу предсказания непрерывного числового значения y на основе признаков. Теперь переходим к задаче **классификации**, где цель совсем другая: не предсказать число, а определить, к какому из заранее заданных классов относится объект.

Переход от регрессии к классификации:

В задаче регрессии мы искали зависимость вида:

$$y = w_0 + w_1 x_1 + w_2 x_2 + \cdots + w_n x_n$$

где y — произвольное число (например, цена квартиры, температура).

В задаче **бинарной классификации** целевая переменная y принимает только два значения: 0 или 1 (например, «спам/не спам», «болен/здоров», «мужчина/женщина»). Мы больше не можем напрямую использовать линейную комбинацию признаков как ответ, потому что она может давать любые значения, не обязательно 0 или 1.

Идея линейного классификатора:

Вместо предсказания числа y мы будем использовать линейную комбинацию признаков для построения **разделяющей границы** между классами. В двумерном случае (два признака x_1 и x_2) эта граница задаётся уравнением:

$$w_0 + w_1 x_1 + w_2 x_2 = 0$$

Это уравнение прямой на плоскости! Все точки, для которых эта линейная комбинация положительна, мы отнесём к одному классу (скажем, класс 1), а все точки с отрицательным значением — к другому классу (класс 0).

Определение 5.19. Линейный классификатор

Линейный классификатор — это алгоритм классификации, который принимает решение о принадлежности объекта к классу на основе знака линейной комбинации его признаков:

$$f(x) = w_0 + w_1 x_1 + w_2 x_2 + \cdots + w_n x_n = w^T x$$

где $w = (w_0, w_1, \dots, w_n)$ — вектор параметров (весов), $x = (1, x_1, \dots, x_n)$ — вектор признаков (добавлена константа 1 для свободного члена w_0).

Решающее правило:

$$\hat{y} = \begin{cases} 1, & \text{если } w^T x \geq 0 \\ 0, & \text{если } w^T x < 0 \end{cases}$$

Почему уравнение теперь выглядит как $w_0 + w_1 x_1 + w_2 x_2 = 0$?

В задаче регрессии мы искали зависимость $y = w^T x$, где y — это предсказываемое значение. В классификации мы не предсказываем конкретное число, а определяем, по какую

сторону от разделяющей гиперплоскости лежит объект. Уравнение $w^T x = 0$ задаёт саму эту гиперплоскость (в 2D — прямую, в 3D — плоскость, в многомерном пространстве — гиперплоскость).

Геометрически:

- Если $w^T x > 0$, точка лежит по одну сторону от гиперплоскости $\Rightarrow$ класс 1
- Если $w^T x < 0$, точка лежит по другую сторону $\Rightarrow$ класс 0
- Если $w^T x = 0$, точка находится точно на границе

Пример в двумерном пространстве:

Рассмотрим задачу классификации на плоскости с двумя признаками x_1 и x_2. Линейный классификатор ищет прямую, которая разделяет точки двух классов. Эта прямая задаётся уравнением:

$$w_0 + w_1 x_1 + w_2 x_2 = 0$$

Коэффициенты w_1 и w_2 определяют наклон прямой, а w_0 — её сдвиг относительно начала координат. Если мы найдём подходящие веса w_0, w_1, w_2, то сможем классифицировать любую новую точку, просто подставив её координаты в формулу и проверив знак результата.

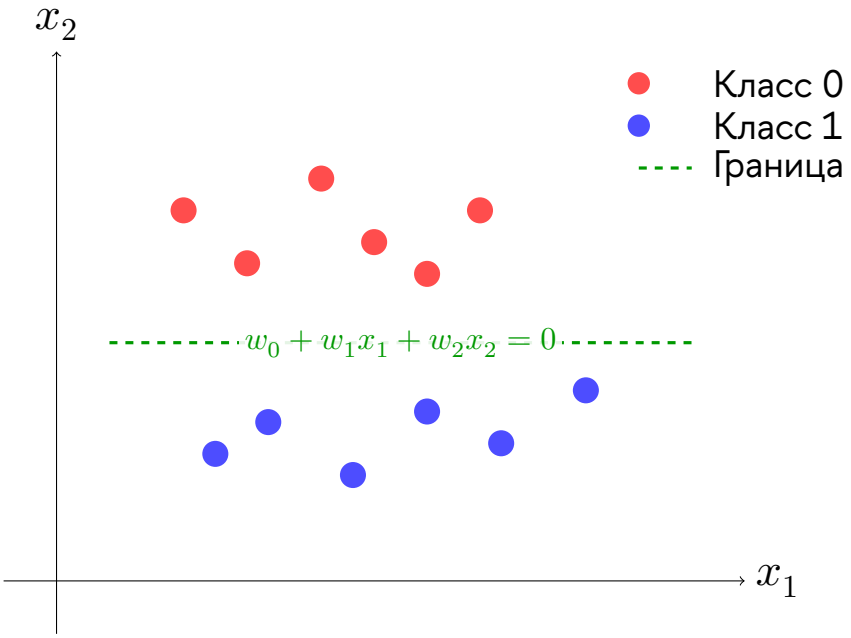


Рис. 8: Линейная классификация: разделяющая прямая делит пространство на два класса

Пример 5.5. Числовой пример

Пусть у нас два признака: x_1 (рост в см) и x_2 (вес в кг), и мы хотим классифицировать людей на мужчин (класс 1) и женщин (класс 0).
Допустим, обученный классификатор имеет параметры: $w_0 = -200$, $w_1 = 1.5$, $w_2 = 0.5$.
Тогда решающее правило:

$$f(x_1, x_2) = -200 + 1.5x_1 + 0.5x_2$$

Для человека ростом 170 см и весом 70 кг:

$$f(170, 70) = -200 + 1.5 \cdot 170 + 0.5 \cdot 70 = -200 + 255 + 35 = 90 > 0$$

Результат положительный $\Rightarrow$ классифицируем как мужчину (класс 1).

Для человека ростом 160 см и весом 50 кг:

$$f(160, 50) = -200 + 1.5 \cdot 160 + 0.5 \cdot 50 = -200 + 240 + 25 = 65 > 0$$

Результат тоже положительный $\Rightarrow$ класс 1.
Конечно, это упрощённый пример!

Определение 5.20. Линейная разделимость

Выборка называется **линейно разделимой**, если существует гиперплоскость, которая идеально разделяет все объекты двух классов без ошибок. Иными словами, можно найти такие веса w, что для всех объектов класса 1 выполняется $w^T x > 0$, а для всех объектов класса 0 выполняется $w^T x < 0$.

Многомерный случай:

Идея обобщается на любое число признаков. Если у нас n признаков, то разделяющая поверхность — это $(n-1)$-мерная гиперплоскость в n-мерном пространстве, задаваемая уравнением:

$$w_0 + w_1 x_1 + w_2 x_2 + \cdots + w_n x_n = 0$$

Все объекты с одной стороны гиперплоскости относятся к классу 1, с другой — к классу 0.

Замечание 5. Ограничения линейных моделей классификации

Линейный классификатор может построить только **линейную разделяющую поверхность**. Если классы устроены сложнее (например, один класс окружает другой, или классы расположены по диагоналям, как в задаче XOR), то одна прямая или гиперплоскость не сможет их идеально разделить.
Проблема XOR: Классическая задача, где нужно разделить четыре точки: $(0, 0)$ и $(1, 1)$ — класс 1, $(0, 1)$ и $(1, 0)$ — класс 0. Никакая прямая не может это сделать без ошибок.
Решения проблемы линейной неразделимости:

- Расширить признаковое пространство (добавить полиномиальные признаки, например $x_1 \cdot x_2$)
- Использовать нелинейные модели
- Применить ансамбли моделей

С этими методами в дальнейшем мы познакомимся ближе.

Обучение линейного классификатора:

Аналогично линейной регрессии, параметры w подбираются путём минимизации функции потерь на обучающей выборке. Однако в отличие от регрессии, где мы минимизировали MSE, в классификации используются другие функции потерь (например, логистическая функция потерь или hinge loss), так как задача дискретная.

Подытожим:

Принципиальное отличие задач заключается в следующем: в регрессии линейная комбинация — это *ответ*, в классификации — это *критерий для разделения*. Мы используем ту же математическую конструкцию (линейная комбинация признаков), но интерпретируем её по-другому в зависимости от типа задачи.

6. Производные, градиенты. Градиентные спуски

6.1. Производная

Определение 6.1. Производная

Производная функции одной переменной $f(x)$ в точке x_0 характеризует скорость изменения функции в этой точке. Геометрически производная — это угловой коэффициент касательной к графику функции. Математически производная определяется как предел:

$$f'(x_0) = \lim_{h \to 0} \frac{f(x_0 + h) - f(x_0)}{h}$$

Если этот предел существует и конечен, функция называется дифференцируемой в точке x_0.

Производная показывает, как функция изменяется при малом изменении аргумента. Если $f'(x_0) > 0$, функция возрастает в этой точке (движение вправо увеличивает функцию). Если $f'(x_0) < 0$, функция убывает (движение вправо уменьшает функцию). Если $f'(x_0) = 0$, точка является стационарной — это может быть локальный минимум, максимум или точка перегиба.

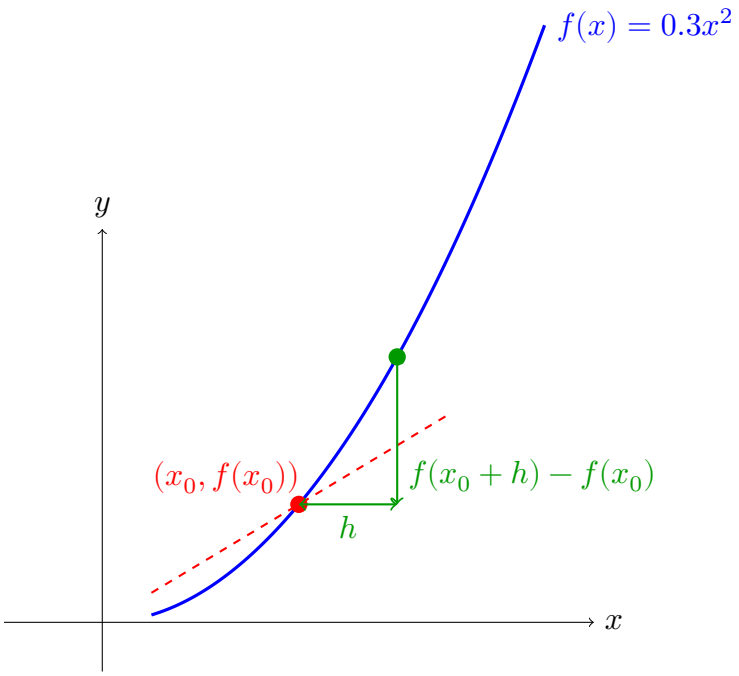


Рис. 9: Геометрический смысл производной: угловой коэффициент касательной

6.2. Таблица производных и свойства

Вычисление производных от базовых функций выполняется по известным формулам. Приведём основные из них:

Таблица производных элементарных функций:

- Константа: $(c)' = 0$
- Степенная функция: $(x^n)' = n \cdot x^{n-1}$
- Корень: $(\sqrt{x})' = \frac{1}{2\sqrt{x}}$
- Показательная функция: $(a^x)' = a^x \ln a$
- Экспоненциальная функция: $(e^x)' = e^x$
- Логарифм: $(\log_a x)' = \frac{1}{x \ln a}$
- Натуральный логарифм: $(\ln x)' = \frac{1}{x}$
- Синус: $(\sin x)' = \cos x$
- Косинус: $(\cos x)' = -\sin x$
- Тангенс: $(\operatorname{tg} x)' = \frac{1}{\cos^2 x}$
- Котангенс: $(\operatorname{ctg} x)' = -\frac{1}{\sin^2 x}$
- Арксинус: $(\arcsin x)' = \frac{1}{\sqrt{1-x^2}}$
- Арккосинус: $(\arccos x)' = -\frac{1}{\sqrt{1-x^2}}$
- Арктангенс: $(\operatorname{arctg} x)' = \frac{1}{1+x^2}$
- Арккотангенс: $(\operatorname{arcctg} x)' = -\frac{1}{1+x^2}$

Свойства производной (правила дифференцирования):

Правило суммы: $(f(x) + g(x))' = f'(x) + g'(x)$ — производная суммы равна сумме производных.

Правило произведения на константу: $(c \cdot f(x))' = c \cdot f'(x)$ — константу можно выносить за знак производной.

Правило произведения: $(f(x) \cdot g(x))' = f'(x) \cdot g(x) + f(x) \cdot g'(x)$ — при дифференцировании произведения каждый множитель дифференцируется, а другой остаётся неизменным, результаты складываются.

Правило частного: $\left(\frac{f(x)}{g(x)}\right)' = \frac{f'(x) \cdot g(x) - f(x) \cdot g'(x)}{(g(x))^2}$ — числитель содержит разность производных с перекрёстным умножением, знаменатель — квадрат исходного знаменателя.

Цепное правило (правило дифференцирования сложной функции): если $h(x) = f(g(x))$, то $h'(x) = f'(g(x)) \cdot g'(x)$.

Пример 6.1. Применение правила суммы

Найдём производную функции $f(x) = 3x^2 + 5x - 7$:

$$f'(x) = (3x^2)' + (5x)' - (7)' = 3 \cdot 2x + 5 \cdot 1 - 0 = 6x + 5$$

Пример 6.2. Применение правила произведения

Найдём производную функции $f(x) = x^2 \cdot e^x$:

$$f'(x) = (x^2)' \cdot e^x + x^2 \cdot (e^x)' = 2x \cdot e^x + x^2 \cdot e^x = e^x(2x + x^2)$$

Пример 6.3. Применение правила частного

Найдём производную функции $f(x) = \frac{x^2}{x+1}$:

$$f'(x) = \frac{(x^2)' \cdot (x+1) - x^2 \cdot (x+1)'}{(x+1)^2} = \frac{2x(x+1) - x^2 \cdot 1}{(x+1)^2}$$

$$= \frac{2x^2 + 2x - x^2}{(x+1)^2} = \frac{x^2 + 2x}{(x+1)^2}$$

Пример 6.4. Применение цепного правила

Найдём производную функции $f(x) = \sin(3x^2 + 1)$:

$$f'(x) = \cos(3x^2 + 1) \cdot (3x^2 + 1)' = \cos(3x^2 + 1) \cdot 6x = 6x \cos(3x^2 + 1)$$

Здесь внешняя функция $\sin(u)$ даёт производную $\cos(u)$, а внутренняя функция $u = 3x^2 + 1$ даёт производную $6x$.

6.3. Частные производные

Частная производная функции $f(x_1, x_2, \ldots, x_n)$ по переменной x_i обозначается $\frac{\partial f}{\partial x_i}$ и вычисляется путём дифференцирования функции по x_i, при этом все остальные переменные рассматриваются как константы:

$$\frac{\partial f}{\partial x_i} = \lim_{h \to 0} \frac{f(x_1, \ldots, x_i + h, \ldots, x_n) - f(x_1, \ldots, x_i, \ldots, x_n)}{h}$$

Частная производная показывает, как быстро изменяется функция в направлении оси x_i. Геометрически это угловой коэффициент касательной к графику функции в направлении, параллельном оси x_i.

Пример 6.5. Вычисление частных производных

Для функции $f(x, y) = x^2 + 3xy + y^2$ найдём частные производные:
Частная производная по x (считаем y константой):

$$\frac{\partial f}{\partial x} = \frac{\partial}{\partial x}(x^2 + 3xy + y^2) = 2x + 3y + 0 = 2x + 3y$$

Частная производная по y (считаем x константой):

$$\frac{\partial f}{\partial y} = \frac{\partial}{\partial y}(x^2 + 3xy + y^2) = 0 + 3x + 2y = 3x + 2y$$

Пример 6.6. Частные производные сложной функции

Для функции $f(x, y) = e^{x^2+y^2}$ найдём частные производные:

По x:

$$\frac{\partial f}{\partial x} = e^{x^2+y^2} \cdot \frac{\partial}{\partial x}(x^2 + y^2) = e^{x^2+y^2} \cdot 2x = 2xe^{x^2+y^2}$$

По y:

$$\frac{\partial f}{\partial y} = e^{x^2+y^2} \cdot \frac{\partial}{\partial y}(x^2 + y^2) = e^{x^2+y^2} \cdot 2y = 2ye^{x^2+y^2}$$

6.4. Градиент

Определение 6.2. Определение градиента

Градиент функции $f(x_1, x_2, \dots, x_n)$ — это вектор, компоненты которого равны частным производным функции по каждой переменной:

$$\text{grad } f = \nabla f = \begin{pmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \\ \vdots \\ \frac{\partial f}{\partial x_n} \end{pmatrix}$$

Обозначается символом ∇ (оператором набла) или grad f.

Геометрический смысл

Градиент указывает направление наискорейшего возрастания функции. Если вы находитесь в точке и хотите максимально быстро увеличить значение функции, идите в направлении вектора градиента. Противоположное направление ($-\nabla f$) показывает путь к максимально быстрому убыванию функции, оно же называется направлением в сторону антиградиента.

Длина вектора градиента показывает крутизну поверхности: чем больше норма градиента, тем быстрее меняется функция.

Градиент для функции одной переменной:

Для функции $f(x) = x^2$:

$$\nabla f = \frac{df}{dx} = 2x$$

В точке $x = 3$ градиент равен 6 — это означает, что функция возрастает в сторону положительных x со скоростью 6 единиц на единицу изменения x.

Градиент для функции двух переменных:

Для функции $f(x, y) = x^2 + y^2$:

$$\nabla f = \begin{pmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \end{pmatrix} = \begin{pmatrix} 2x \\ 2y \end{pmatrix}$$

В точке $(1, 2)$ градиент равен $\begin{pmatrix} 2 \\ 4 \end{pmatrix}$ — это вектор, указывающий направление максимального увеличения функции.

Ключевые свойства

1. Направление наибыстрейшего роста.

Единственное направление, в котором функция возрастает быстрее всего — это направление градиента.

2. Нулевой градиент в стационарных точках.

Если $\nabla f = 0$, то точка является критической (может быть минимумом, максимумом или седловой точкой). В этих точках локальное поведение функции определяется второй производной (матрицей Гессе).

3. Инвариантность.

Геометрический смысл градиента не зависит от выбора системы координат — это инвариантный объект, который существует независимо от того, какие переменные мы используем для описания функции.

6.5. Градиентный спуск

Градиентный спуск — это итеративный алгоритм оптимизации, используемый для нахождения минимума функции. Идея проста: начинаем с некоторой начальной точки w_0 и шаг за шагом движемся в направлении, противоположном градиенту, то есть в направлении наискорейшего убывания функции.

Классическая реализация градиентного спуска обновляет параметры по следующему правилу:

$$w_{t+1} = w_t - \eta \nabla Q(w_t)$$

где w_t — вектор параметров на шаге t, η — коэффициент обучения (скорость обучения), $\nabla Q(w_t)$ — градиент функции потерь в точке w_t, а индекс $t+1$ обозначает следующее значение параметров.

На каждой итерации алгоритм вычисляет градиент функции потерь в текущей точке, умножает его на коэффициент обучения η и вычитает из текущих параметров. Знак минус перед градиентом критичен — он обеспечивает движение в направлении убывания функции.

Коэффициент обучения η контролирует размер шага. Если η слишком мал, алгоритм сходится медленно и требует много итераций. Если η слишком велик, алгоритм может «перепрыгнуть» через минимум и начать расходиться. Выбор оптимального η — это практическая задача, требующая экспериментирования.

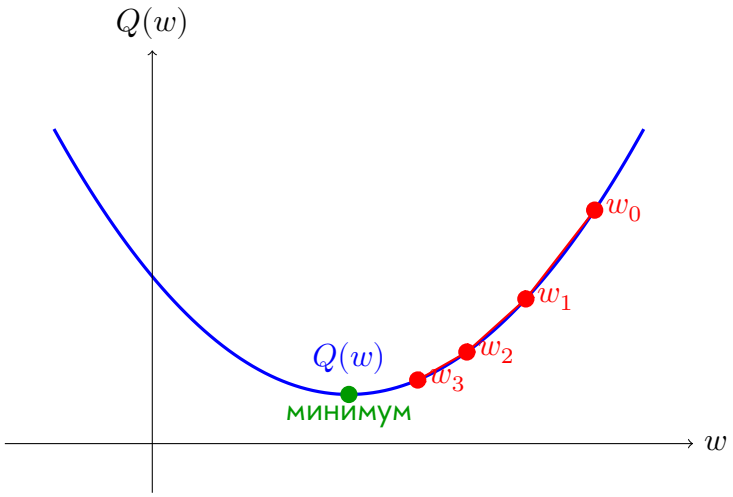


Рис. 10: Градиентный спуск: итеративное движение к минимуму функции потерь

Пример 6.7. Градиентный спуск для функции одной переменной

Рассмотрим функцию $Q(w) = (w-3)^2$. Её минимум в точке $w = 3$.
Градиент: $\nabla Q(w) = \frac{dQ}{dw} = 2(w-3)$
Начнём с $w_0 = 0$ и $\eta = 0.1$:

Итерация 1:

$$w_1 = w_0 - \eta \nabla Q(w_0) = 0 - 0.1 \cdot 2(0 - 3) = 0 - 0.1 \cdot (-6) = 0.6$$

Итерация 2:

$$w_2 = w_1 - \eta \nabla Q(w_1) = 0.6 - 0.1 \cdot 2(0.6 - 3) = 0.6 - 0.1 \cdot (-4.8) = 1.08$$

Итерация 3:

$$w_3 = w_2 - \eta \nabla Q(w_2) = 1.08 - 0.1 \cdot 2(1.08 - 3) = 1.08 + 0.384 = 1.464$$

Продолжая итерации, значение w будет приближаться к оптимуму $w^* = 3$.

Пример 6.8. Градиентный спуск для функции двух переменных

Рассмотрим функцию $Q(x, y) = x^2 + y^2$. Её минимум в точке $(0, 0)$.
Градиент: $\nabla Q = (2x, 2y)$
Начнём с $(x_0, y_0) = (4, 3)$ и $\eta = 0.1$:
Итерация 1:

$$\begin{pmatrix} x_1 \\ y_1 \end{pmatrix} = \begin{pmatrix} x_0 \\ y_0 \end{pmatrix} - \eta \nabla Q(x_0, y_0) = \begin{pmatrix} 4 \\ 3 \end{pmatrix} - 0.1 \begin{pmatrix} 8 \\ 6 \end{pmatrix} = \begin{pmatrix} 3.2 \\ 2.4 \end{pmatrix}$$

Итерация 2:

$$\begin{pmatrix} x_2 \\ y_2 \end{pmatrix} = \begin{pmatrix} 3.2 \\ 2.4 \end{pmatrix} - 0.1 \begin{pmatrix} 6.4 \\ 4.8 \end{pmatrix} = \begin{pmatrix} 2.56 \\ 1.92 \end{pmatrix}$$

С каждой итерацией точка движется к минимуму $(0, 0)$.

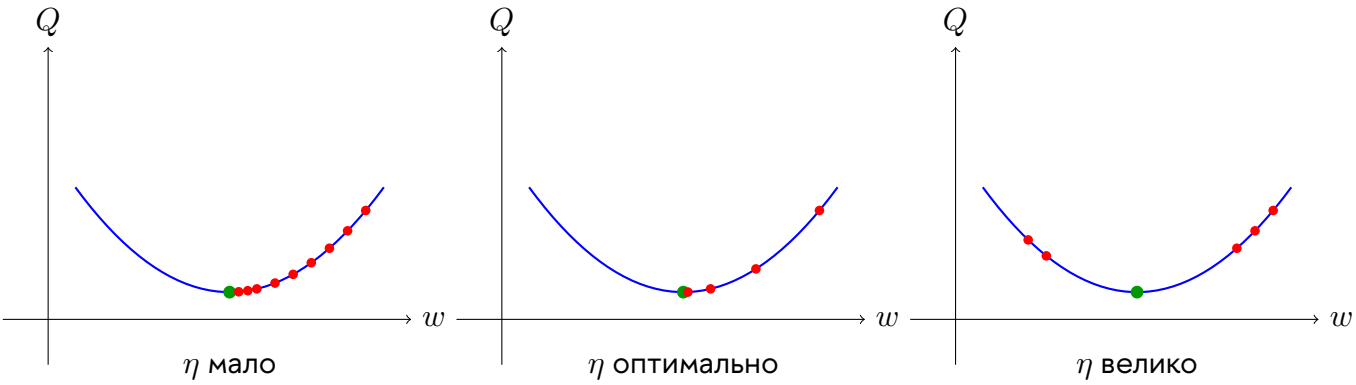


Рис. 11: Влияние коэффициента обучения на сходимость градиентного спуска

Процесс останавливается, когда выполнено одно из условий: градиент становится близким к нулю (достигнута стационарная точка), изменение параметров между итерациями становится меньше заданного порога, или достигнут максимальный допустимый номер итераций. На практике последнее условие используется чаще всего для контроля вычислительных затрат.

Преимущества классического градиентного спуска состоят в его простоте понимания и реализации. Однако при работе с большими наборами данных алгоритм становится медленным, так как на каждой итерации требуется вычисление градиента по всей выборке, что для миллионов примеров вычислительно дорого.

6.6. Стохастический градиентный спуск

Как мы видели, вычисление полного градиента $\nabla Q(w)$ требует обработки всей выборки. При больших данных это становится неприемлемо медленным. Выход — заменить истинный градиент на псевдоградиент (субградиент), вычисляемый на основе одного или нескольких случайно выбранных примеров.

Ключевое требование к псевдоградиенту $\nabla \tilde{Q}(w_t)$: его направление должно в среднем образовывать острый угол с истинным градиентом $\nabla Q(w)$. Это гарантирует, что в среднем мы движемся к точке минимума.

Самый простой вариант — на каждой итерации случайно выбирать один объект k из выборки и вычислять градиент только по нему:

$$\nabla \tilde{Q}(w_t) = \nabla L_k(w_t, x_k)$$

По закону больших чисел, при большом количестве итераций этот псевдоградиент сходится к истинному градиенту, и алгоритм достигает точки минимума.

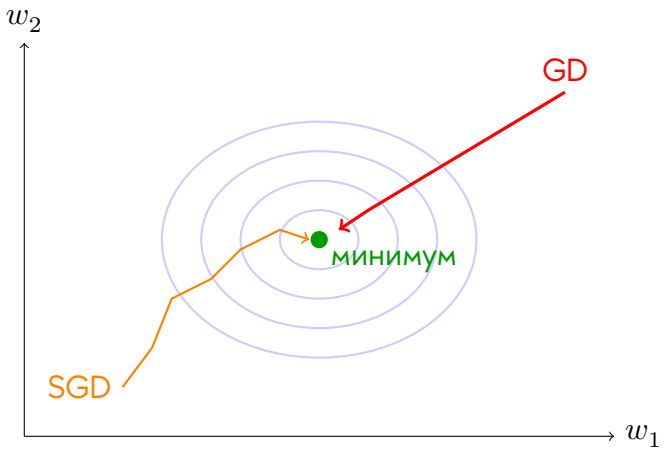


Рис. 12: Сравнение траекторий: классический градиентный спуск (гладкая) и стохастический (зигзагообразная с шумом)

Алгоритм SGD:

1. Инициализация весов w_0 начальными значениями.
2. Начальное вычисление функционала качества: $\bar{Q} = \frac{1}{l}\sum_{i=1}^{l} L_i(w)$.
3. Цикл (пока не достигнуто условие остановки):
   - Случайный выбор наблюдения x_k из обучающей выборки.
   - Вычисление ошибки: $\varepsilon_k = L_k(w)$.
   - Шаг градиентного спуска: $w_{t+1} = w_t - \eta \nabla L(w)$.
   - Пересчёт функционала качества (экспоненциальное скользящее среднее): $\bar{Q} = \lambda \varepsilon_k + (1-\lambda)\bar{Q}$.
4. Остановка при достижении заданного качества или числа итераций.

Вывод формулы экспоненциального скользящего среднего

Здесь у вас может вызвать недоумение формула пересчёта среднего показателя качества $\bar{Q} = \lambda \varepsilon_n + (1-\lambda)\bar{Q}$. Что это такое и зачем она нужна? Смотрите, в процессе обучения

алгоритма нам нужно уметь оценивать качество найденных весовых коэффициентов, а это есть функционал $Q(w)$. Например, когда он достигает определённого значения, то процесс обучения можно остановить. Но, формально, он вычисляется как среднее арифметическое от функций потерь по всей выборке. А это снова большой объём вычислений, которого, как раз, хотим избежать. Поэтому в алгоритмах SGD функционал $\bar{Q}$ вычисляется на основе экспоненциального среднего.

Откуда взялась эта рекуррентная формула? Давайте представим, что мы вычисляем простое арифметическое среднее по m величинам:

$$\bar{Q}_m = \frac{1}{m}\varepsilon_m + \frac{1}{m}\varepsilon_{m-1} + \frac{1}{m}\varepsilon_{m-2} + ... + \frac{1}{m}\varepsilon_1$$

Можно заметить, что:

$$m \cdot \bar{Q}_m - \varepsilon_m = \varepsilon_{m-1} + \varepsilon_{m-2} + ... + \varepsilon_1$$

$$\bar{Q}_{m-1} = \frac{1}{m-1}\left[m \cdot \bar{Q}_m - \varepsilon_m\right] = \frac{m}{m-1}\bar{Q}_m - \frac{1}{m-1}\varepsilon_m$$

откуда

$$\bar{Q}_m = \frac{m-1}{m}\bar{Q}_{m-1} + \frac{m-1}{m} \cdot \frac{1}{m-1} \cdot \varepsilon_m$$

$$\bar{Q}_m = \frac{1}{m}\varepsilon_m + \left(1 - \frac{1}{m}\right)\bar{Q}_{m-1}$$

Вот формула рекуррентного пересчёта среднего арифметического при появлении нового слагаемого ε_m. Экспоненциальное скользящее среднее работает по тому же принципу, только вместо изменяющегося шага $1/m$ берётся постоянная величина λ:

$$\bar{Q}_m = \lambda\varepsilon_m + (1-\lambda)\bar{Q}_{m-1}$$

Если расписать эту формулу, то она будет эквивалентна следующей сумме:

$$\bar{Q}_m = \lambda\varepsilon_m + (1-\lambda)\lambda\varepsilon_{m-1} + (1-\lambda)^2\lambda\varepsilon_{m-2} + ...$$

Мы здесь видим экспоненциально убывающие коэффициенты при ε_i. Значение параметра λ принято подбирать в соответствии с правилом:

$$\lambda = \frac{2}{N+1}$$

где N — интервал сглаживания. Например, при $N = 99$ получаем $\lambda = 0.02$.

7. Случайные величины в теории вероятностей

7.1. Определение случайной величины

Определение 7.1. Определение случайной величины

Случайной величиной называется величина, которая в результате испытания принимает одно и только одно значение, заранее не известное и зависящее от случайных причин.

Случайные величины обозначаются прописными буквами: X, Y, Z, а их конкретные значения — строчными: x, y, z.

Пример 1. Подбрасываем игральный кубик. Количество выпавших очков — случайная величина X. Возможные значения: $1, 2, 3, 4, 5, 6$.

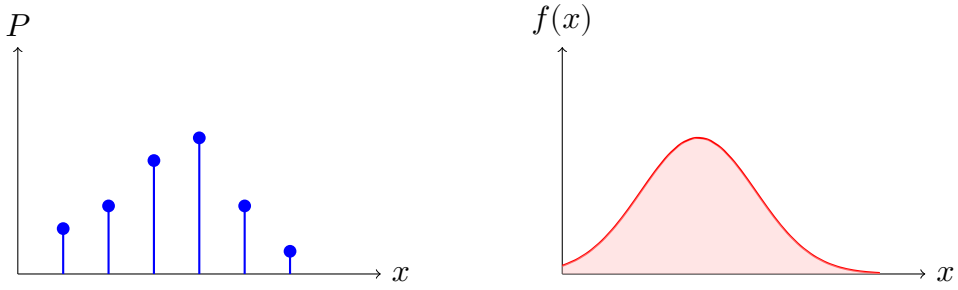
Пример 2. Дальность полёта снаряда — случайная величина, так как на неё влияет множество случайных факторов: сила и направление ветра, отклонения в массе пороха и т.д.

7.2. Дискретные и непрерывные случайные величины

Определение 7.2. Типы случайных величин

Дискретной называется случайная величина, которая принимает отдельные, изолированные значения. Число возможных значений конечно или счётно.

Непрерывной называется случайная величина, которая может принимать любые значения из некоторого промежутка (возможно, бесконечного).



Дискретная СВ Непрерывная СВ

Рис. 13: Сравнение дискретной и непрерывной случайных величин

7.3. Закон распределения дискретной случайной величины

Дискретную случайную величину удобно задавать с помощью таблицы распределения:

X	x_1	x_2	...	x_n
P	p_1	p_2	...	p_n

При этом сумма всех вероятностей равна единице:

$$\sum_{i=1}^{n} p_i = 1$$

Пример. Бросание игрального кубика:

X	1	2	3	4	5	6
P	$\frac{1}{6}$	$\frac{1}{6}$	$\frac{1}{6}$	$\frac{1}{6}$	$\frac{1}{6}$	$\frac{1}{6}$

7.4. Математическое ожидание

Математическое ожидание (среднее значение) случайной величины — это взвешенное среднее всех её возможных значений.

Для **дискретной** случайной величины:

$$E[X] = M[X] = \sum_{i=1}^{n} x_i \cdot p_i$$

Вероятностный смысл: если проводить эксперимент много раз, то среднее арифметическое полученных значений будет стремиться к математическому ожиданию (закон больших чисел).

Пример. Найдём математическое ожидание числа очков при броске кубика:

$$E[X] = 1 \cdot \frac{1}{6} + 2 \cdot \frac{1}{6} + 3 \cdot \frac{1}{6} + 4 \cdot \frac{1}{6} + 5 \cdot \frac{1}{6} + 6 \cdot \frac{1}{6} = \frac{21}{6} = 3.5$$

Свойство 7.1. Свойства математического ожидания

- $E[c] = c$
- $E[cX] = c \cdot E[X]$
- $E[X + Y] = E[X] + E[Y]$
- $E[X \cdot Y] = E[X] \cdot E[Y]$
- $E[aX + bY + c] = a \cdot E[X] + b \cdot E[Y] + c$

7.5. Дисперсия

Определение 7.3. Определение дисперсии

Дисперсия — это мера разброса случайной величины относительно её математического ожидания.

$$D[X] = \mathrm{Var}[X] = E\left[(X - E[X])^2\right]$$

Дисперсия всегда неотрицательна и измеряется в квадратах единиц исходной величины.

Вычислительная формула (удобна для расчётов):

$$D[X] = E[X^2] - (E[X])^2$$

Для **дискретной** случайной величины:

$$D[X] = \sum_{i=1}^{n} (x_i - E[X])^2 \cdot p_i = \sum_{i=1}^{n} x_i^2 \cdot p_i - (E[X])^2$$

Свойство 7.2. Свойства дисперсии

- $D[c] = 0$
- $D[cX] = c^2 D[X]$
- $D[X + c] = D[X]$
- $D[X + Y] = D[X] + D[Y]$ (если X и Y независимы)

7.6. Среднеквадратическое отклонение

Определение 7.4. Определение СКО

Среднеквадратическое (стандартное) отклонение — корень из дисперсии:

$$\sigma_X = \sqrt{D[X]}$$

Пример. Продолжим пример с кубиком. Найдём дисперсию:
Сначала найдём $E[X^2]$:

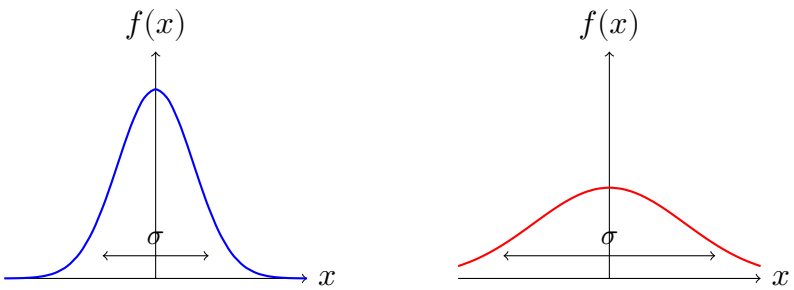
$$E[X^2] = 1^2 \cdot \frac{1}{6} + 2^2 \cdot \frac{1}{6} + 3^2 \cdot \frac{1}{6} + 4^2 \cdot \frac{1}{6} + 5^2 \cdot \frac{1}{6} + 6^2 \cdot \frac{1}{6} = \frac{91}{6}$$

Дисперсия:

$$D[X] = E[X^2] - (E[X])^2 = \frac{91}{6} - \left(\frac{7}{2}\right)^2 = \frac{91}{6} - \frac{49}{4} = \frac{35}{12} \approx 2.92$$

Стандартное отклонение:

$$\sigma_X = \sqrt{\frac{35}{12}} \approx 1.71$$



Малая дисперсия Большая дисперсия

Рис. 14: Сравнение распределений с разной дисперсией

7.7. Биномиальное распределение

Случайная величина X имеет **биномиальное распределение** с параметрами n и p, если:

$$P(X = k) = C_n^k p^k (1-p)^{n-k}, \quad k = 0, 1, \ldots, n$$

где $C_n^k = \frac{n!}{k!(n-k)!}$ — биномиальный коэффициент.
Обозначение: $X \sim \mathrm{Bin}(n, p)$
Характеристики:

$$E[X] = np, \quad D[X] = np(1-p)$$

7.8. Нормальное распределение

Случайная величина X имеет **нормальное распределение** с параметрами μ и σ^2, если её плотность:

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$

Обозначение: $X \sim N(\mu, \sigma^2)$
Характеристики:

$$E[X] = \mu, \quad D[X] = \sigma^2$$

Правило трёх сигм: для нормального распределения примерно 99.7% значений лежат в интервале $(\mu - 3\sigma, \mu + 3\sigma)$.

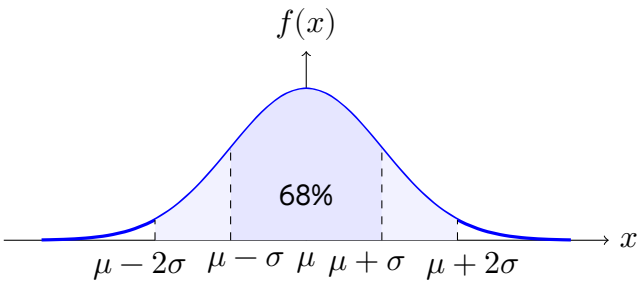


Рис. 15: Нормальное распределение (кривая Гаусса)

7.9. Неравенство Чебышёва

Неравенство Чебышёва. Вероятность того, что отклонение случайной величины X от ее математического ожидания по абсолютной величине меньше положительного числа ε, не меньше чем $1 - \frac{D[X]}{\varepsilon^2}$.

$$P(|X - E[X]| \leq \varepsilon) \geq 1 - \frac{D[X]}{\varepsilon^2}$$

8. NumPy: Фундаментальные концепции

8.1. Введение и история

NumPy (Numerical Python) — фундаментальная библиотека для научных вычислений на Python. Создана в 2006 году Трэвисом Олифантом как обёртка над быстрыми C и Fortran-библиотеками. Основная цель — предоставить эффективный интерфейс для работы с многомерными массивами и математическими функциями.

8.2. Преимущества NumPy перед списками Python

1. **Производительность:** Операции работают в 10–100 раз быстрее благодаря реализации на C.
2. **Удобство:** Векторизованные операции позволяют избежать явных циклов.
3. **Память:** NumPy-массивы занимают значительно меньше памяти.
4. **Функциональность:** Встроенные математические, статистические и линейно-алгебраические функции.

8.3. Объект ndarray и его атрибуты

Центральная структура NumPy — n-мерный массив (`ndarray`). Основные атрибуты:

- **shape** — кортеж размеров массива в каждом измерении
- **dtype** — тип данных элементов (`int32`, `float64`, `complex128`)
- **ndim** — количество измерений (размерность)
- **size** — общее количество элементов: $\prod(\text{shape})$
- **itemsize** — размер одного элемента в байтах

1) Вывод атрибутов массива:

```python
import numpy as np

a = np.array([[1, 2, 3], [4, 5, 6]])
print(f"Shape: {a.shape}")        # (2, 3)
print(f"DType: {a.dtype}")        # int64
print(f"NDim: {a.ndim}")          # 2
print(f"Size: {a.size}")          # 6
print(f"Itemsize: {a.itemsize}") # 8
```

8.4. Создание массивов

8.4.1 Из списков Python

2) Создание 1D и 2D массивов из списков:

```python
a = np.array([1, 2, 3])           # 1D массив
b = np.array([[1, 2], [3, 4]])    # 2D массив (матрица)
```

8.4.2 Встроенные функции для создания массивов

3) Создание специальных массивов:

```
zeros = np.zeros((3, 3))     # Матрица 3x3 из нулей
ones = np.ones(5)            # Вектор из 5 единиц
full = np.full((2, 2), 7)    # Матрица 2x2, заполненная числом 7
eye = np.eye(3)              # Единичная матрица 3x3
```

8.4.3 Последовательности

4) Создание арифметических и логарифмических последовательностей:

```
a = np.arange(0, 10, 2)      # [0, 2, 4, 6, 8]
b = np.linspace(0, 1, 5)     # [0., 0.25, 0.5, 0.75, 1.]
c = np.logspace(0, 2, 4)     # [1., 10., 100., 1000.]
```

8.4.4 Случайные значения

5) Генерация случайных массивов:

```
a = np.random.rand(3, 3)          # Uniform [0, 1), размер 3x3
b = np.random.randn(4)            # Нормальное N(0,1), размер 4
c = np.random.randint(1, 10, 5) # Целые из [1, 10), размер 5
d = np.random.choice([1, 2, 3], 5)  # Выборка с повторениями
```

8.5. Индексация и срезы

8.5.1 Одномерные массивы

6) Индексация и срезы в 1D массиве:

```
a = np.array([10, 20, 30, 40, 50])
print(a[0])      # 10 (первый элемент)
print(a[-1])     # 50 (последний элемент)
print(a[1:4])    # [20, 30, 40] (срез)
print(a[::2])    # [10, 30, 50] (каждый второй)
```

8.5.2 Многомерные массивы

7) Индексация и срезы в матрице:

```
a = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print(a[0, 0])       # 1 (элемент [0][0])
print(a[1, :])       # [4, 5, 6] (вторая строка)
print(a[:, 1])       # [2, 5, 8] (второй столбец)
print(a[0:2, 1:3])   # [[2, 3], [5, 6]] (подматрица)
```

8.5.3 Булева индексация

8) Фильтрация элементов по условию:

```
a = np.array([1, 5, 3, 8, 2, 9])
mask = a > 4
print(a[mask])                    # [5, 8, 9]
result = a[(a > 2) & (a < 8)]     # [5, 3] (комбинированное условие)
```

8.6. Типы данных (dtype)

9) Явное указание типа и преобразование типов:

```
a = np.array([1, 2, 3], dtype=np.int32)
b = np.array([1.0, 2.0, 3.0], dtype=np.float64)
c = np.array([True, False, True], dtype=bool)

e = a.astype(float)  # int -> float: [1., 2., 3.]
f = b.astype(int)    # float -> int: [1, 2, 3]
```

8.7. Операции и векторизация

Векторизация — замена явных циклов на операции над целыми массивами. Код становится быстрее (работает на C-уровне), понятнее и компактнее.

8.7.1 Поэлементные арифметические операции

10) Арифметические операции над массивами:

```
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])

print(a + b)     # [5, 7, 9]
print(a - b)     # [-3, -3, -3]
print(a * b)     # [4, 10, 18]
print(a / b)     # [0.25, 0.4, 0.5]
print(a ** 2)    # [1, 4, 9]
print(a + 10)    # [11, 12, 13] (со скаляром)
```

8.7.2 Математические функции

11) Применение математических функций к массиву:

```
a = np.array([1, 4, 9, 16])

print(np.sqrt(a))    # [1., 2., 3., 4.] (корень)
print(np.exp(a))     # Экспонента каждого элемента
print(np.log(a))     # Натуральный логарифм
print(np.sin(a))     # Синус каждого элемента
print(np.abs(np.array([-1, 2, -3])))  # [1, 2, 3]
```

8.8. Транслирование (Broadcasting)

Транслирование — механизм, позволяющий выполнять операции над массивами разных форм. NumPy автоматически расширяет меньший массив до совместимой формы.

Правила транслирования:

1. Если массивы имеют разное число измерений, форма меньшего дополняется единицами слева.
2. Два массива совместимы в измерении, если размеры равны или один из них равен 1.
3. При несовместимости возникает ошибка `ValueError`.

12) Примеры broadcasting:

```
a = np.array([[1, 2], [3, 4]])

# Вычитание скаляра
print(a - 1)            # [[0, 1], [2, 3]]

# Вычитание вектора из каждой строки
b = np.array([10, 20])
print(a - b)            # [[-9, -18], [-7, -16]]

# Вычитание вектора-столбца
c = np.array([[10], [20]])
print(a - c)            # [[-9, -8], [-17, -16]]
```

8.9. Изменение формы массива

13) Reshape, flatten, ravel:

```
a = np.arange(12)          # [0, 1, 2, ..., 11]
b = a.reshape(3, 4)        # Матрица 3x4
c = a.reshape(2, 2, 3)     # 3D массив 2x2x3
d = b.flatten()            # В 1D (создает копию)
e = b.ravel()              # В 1D (обычно view)
```

8.10. Операции над осями (axis)

`axis=0` означает операцию вдоль строк (по столбцам), `axis=1` — вдоль столбцов (по строкам).

14) Агрегация по осям:

```
a = np.array([[1, 2, 3], [4, 5, 6]])

print(a.sum())          # 21 (сумма всех)
print(a.sum(axis=0))    # [5, 7, 9] (по столбцам)
print(a.sum(axis=1))    # [6, 15] (по строкам)
print(a.mean(axis=0)) # [2.5, 3.5, 4.5]
print(a.max(axis=1))    # [3, 6]
```

8.11. Объединение и разделение массивов

15) Горизонтальное и вертикальное объединение:

```
a = np.array([[1, 2], [3, 4]])
b = np.array([[5, 6], [7, 8]])

# Горизонтальное (вдоль axis=1)
h = np.hstack([a, b])           # [[1, 2, 5, 6], [3, 4, 7, 8]]
h = np.concatenate([a, b], axis=1)

# Вертикальное (вдоль axis=0)
v = np.vstack([a, b])           # [[1, 2], [3, 4], [5, 6], [7, 8]]
v = np.concatenate([a, b], axis=0)

# Разделение
split_h = np.hsplit(a, 2)
split_v = np.vsplit(v, 2)
```

8.12. Линейная алгебра

16) Скалярное произведение векторов:

```
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])

dot_prod = np.dot(a, b)  # 1*4 + 2*5 + 3*6 = 32
```

17) Матричное умножение:

```
A = np.array([[1, 2], [3, 4]])
B = np.array([[5, 6], [7, 8]])

C = np.dot(A, B)     # Матричное умножение
C = A @ B            # Альтернативный синтаксис (Python 3.5+)
```

18) Определитель, обратная матрица:

```
A = np.array([[1, 2], [3, 4]])

det_A = np.linalg.det(A)        # Определитель
A_inv = np.linalg.inv(A)        # Обратная матрица
```

19) Решение системы линейных уравнений $Ax = b$:

```
A = np.array([[2, 1], [1, 3]])
b = np.array([5, 6])

x = np.linalg.solve(A, b)  # Решение системы
```

8.13. Статистические функции

20) Основные статистические функции:

```
a = np.array([1, 2, 3, 4, 5])

print(np.mean(a))            # 3.0 (среднее)
print(np.median(a))          # 3.0 (медиана)
print(np.std(a))             # Стандартное отклонение
print(np.var(a))             # Дисперсия
print(np.min(a))             # 1
print(np.max(a))             # 5
print(np.sum(a))             # 15
print(np.prod(a))            # 120 (произведение)
print(np.percentile(a, 25)) # 2.0 (25-й процентиль)
```

8.14. Сортировка и поиск

21) Сортировка и поиск элементов:

```
a = np.array([3, 1, 4, 1, 5, 9, 2, 6])

sorted_a = np.sort(a)        # [1, 1, 2, 3, 4, 5, 6, 9]
indices = np.argsort(a)      # Индексы для сортировки
max_idx = np.argmax(a)       # Индекс максимума: 5
min_idx = np.argmin(a)       # Индекс минимума: 1
where = np.where(a > 4)      # Индексы элементов > 4
unique = np.unique(a)        # Уникальные: [1, 2, 3, 4, 5, 6, 9]
```

9. Pandas: Анализ и обработка данных

9.1. Введение в Pandas

Pandas — библиотека для работы с табличными данными. Основные структуры:

- **Series** — 1D массив с метками (индексом)
- **DataFrame** — 2D таблица с метками для строк и столбцов

9.2. Series: одномерная структура

1) Создание Series и доступ к элементам:

```python
import pandas as pd

s = pd.Series([10, 20, 30], index=["a", "b", "c"])
print(s)
# a    10
# b    20
# c    30

print(s["a"])        # 10 (по метке)
print(s.iloc[0])     # 10 (по позиции)
print(s.loc["a"])    # 10 (явно по метке)
print(s + 5)         # [15, 25, 35]
print(s.mean())      # 20.0
```

9.3. DataFrame: двумерная структура

2) Создание DataFrame из словаря:

```python
import pandas as pd

data = {
    "name": ["Alice", "Bob", "Charlie"],
    "age": [25, 30, 35],
    "salary": [50000, 60000, 75000]
}
df = pd.DataFrame(data)

print(df.shape)        # (3, 3)
print(df.columns)      # Index(["name", "age", "salary"])
print(df.dtypes)       # Типы данных столбцов
print(df.head(2))      # Первые 2 строки
print(df.describe())   # Описательная статистика
```

9.4. Выборка данных

3) Выборка столбцов:

```
print(df["name"])            # Series (один столбец)
print(df[["name", "age"]])   # DataFrame (несколько столбцов)
```

4) Выборка строк по позиции (iloc) и по метке (loc):

```
print(df.iloc[0])        # Первая строка
print(df.iloc[0:2])      # Первые 2 строки

df_indexed = df.set_index("name")
print(df_indexed.loc["Alice"])  # Строка с меткой "Alice"
```

5) Булева индексация:

```
print(df[df["age"] > 25])                # Строки, где age > 25
print(df.loc[df["age"] > 25, "name"])    # Только имена старше 25
```

9.5. Добавление и удаление данных

6) Добавление и удаление столбцов:

```
df["bonus"] = df["salary"] * 0.1     # Добавить столбец
df = df.drop("bonus", axis=1)        # Удалить столбец
```

7) Добавление и удаление строк:

```
new_row = pd.DataFrame({"name": ["David"], "age": [40], "salary":
 ↪  [80000]})
df = pd.concat([df, new_row], ignore_index=True)

df = df.drop(0)                      # Удалить строку с индексом 0
df = df.reset_index(drop=True)       # Сбросить индекс
```

9.6. Группировка (Group By)

Группировка следует парадигме **Split-Apply-Combine**: разбить данные на группы по ключу, применить функцию к каждой группе, объединить результаты.

8) Группировка и агрегация:

```
sales_data = {
    "region": ["North", "South", "North", "East", "South"],
    "quarter": ["Q1", "Q1", "Q2", "Q1", "Q2"],
    "revenue": [100, 150, 120, 200, 130]
}
df = pd.DataFrame(sales_data)

# Сумма по регионам
print(df.groupby("region")["revenue"].sum())
# North    220
```

```
# South    280
# East     200

# Несколько агрегирующих функций
print(df.groupby("region")["revenue"].agg(["sum", "mean", "count"]))

# Группировка по нескольким столбцам
print(df.groupby(["region", "quarter"])["revenue"].sum())
```

9.7. Объединение таблиц (Merge)

9) Различные типы join:

```
left = pd.DataFrame({"key": ["A", "B", "C"], "value_left": [1, 2, 3]})
right = pd.DataFrame({"key": ["A", "B", "D"], "value_right": [4, 5, 6]})

# Inner join (только совпадающие ключи: A, B)
inner = pd.merge(left, right, on="key", how="inner")

# Left join (все из левой: A, B, C)
left_join = pd.merge(left, right, on="key", how="left")

# Full outer join (все ключи: A, B, C, D)
outer = pd.merge(left, right, on="key", how="outer")
```

10) Конкатенация (склеивание):

```
vertical = pd.concat([left, right])            # По вертикали
horizontal = pd.concat([left, right], axis=1) # По горизонтали
```

9.8. Работа с пропусками (NaN)

11) Проверка пропусков:

```
df = pd.DataFrame({
    "A": [1, 2, None, 4],
    "B": [5, None, 7, 8],
    "C": [9, 10, 11, 12]
})

print(df.isnull())          # Boolean DataFrame
print(df.isnull().sum())    # Количество NaN в каждом столбце
```

12) Удаление пропусков:

```
df_dropped = df.dropna()              # Удалить строки с NaN
df_dropped = df.dropna(subset=["A"]) # Только где NaN в столбце A
```

13) Заполнение пропусков:

```
df_filled = df.fillna(0)                # Заполнить нулями
df_filled = df.fillna(df.mean())        # Заполнить средними
df_filled = df.fillna(method="ffill") # Forward fill
df_filled = df.fillna(method="bfill") # Backward fill
```

9.9. Применение функций

14) Применение функций к DataFrame:

```
df = pd.DataFrame({
    "A": [1, 2, 3],
    "B": [4, 5, 6]
})

# К каждому элементу
df_doubled = df.map(lambda x: x * 2)   # pandas >= 2.1

# К столбцу
df["C"] = df["A"].apply(lambda x: x ** 2)

# К строке
df["sum_row"] = df.apply(lambda row: row["A"] + row["B"], axis=1)
```

9.10. Сводные таблицы (Pivot Table)

15) Создание сводной таблицы:

```
sales = pd.DataFrame({
    "date": ["2023-01-01", "2023-01-01", "2023-01-02",
     ↪  "2023-01-02"],
    "product": ["A", "B", "A", "B"],
    "quantity": [10, 20, 15, 25]
})

pivot = sales.pivot_table(
    values="quantity",
    index="product",
    columns="date",
    aggfunc="sum"
)
print(pivot)
# date       2023-01-01  2023-01-02
# product
# A                  10          15
# B                  20          25
```

9.11. Работа с временными рядами

16) Создание временного индекса:

```python
import pandas as pd

dates = pd.date_range("2023-01-01", periods=5)
df = pd.DataFrame({"value": [10, 15, 12, 18, 20]}, index=dates)

print(df.loc["2023-01-02"])               # Доступ по дате
print(df.loc["2023-01-02":"2023-01-04"]) # Срез по датам
```

17) Ресэмплинг (изменение частоты):

```
weekly = df.resample("W").sum()      # Еженедельная сумма
monthly = df.resample("M").mean()    # Ежемесячное среднее
```

18) Сдвиг и скользящие функции:

```
df["prev_value"] = df["value"].shift(1)                # Сдвиг на 1
df["rolling_mean"] = df["value"].rolling(window=2).mean()  #
 ↪  Скользящее среднее
df["expanding_sum"] = df["value"].expanding().sum()  # Нарастающая
 ↪  сумма
```

9.12. Сортировка и ранжирование

19) Сортировка DataFrame:

```
df = pd.DataFrame({
    "name": ["Alice", "Bob", "Charlie"],
    "salary": [50000, 60000, 55000]
})

df_sorted = df.sort_values(by="salary", ascending=False)
df_sorted = df.sort_values(by=["salary", "name"])  # По нескольким
df["rank"] = df["salary"].rank(method="average")   # Ранжирование
```

9.13. Полезные приёмы

20) Min-Max нормализация в диапазон $[0, 1]$:

```
a = np.array([10, 20, 30, 40, 50])
normalized = (a - a.min()) / (a.max() - a.min())
# [0.0, 0.25, 0.5, 0.75, 1.0]
```

21) Z-score: $z = \frac{x-\mu}{\sigma}$:

```
a = np.array([10, 20, 30, 40, 50])
z_scores = (a - a.mean()) / a.std()
# [-1.41, -0.71, 0.00, 0.71, 1.41]
```

22) Удаление выбросов методом межквартильного размаха:

```
df = pd.DataFrame({"value": [1, 2, 3, 4, 5, 100]})

Q1 = df["value"].quantile(0.25)
Q3 = df["value"].quantile(0.75)
IQR = Q3 - Q1

lower_bound = Q1 - 1.5 * IQR
upper_bound = Q3 + 1.5 * IQR

df_no_outliers = df[(df["value"] >= lower_bound) &
                    (df["value"] <= upper_bound)]
```

23) Категоризация данных. Разбиение числовых данных на категории (pd.cut):

```
df = pd.DataFrame({
    "name": ["Alice", "Bob", "Charlie", "David"],
    "age": [25, 35, 55, 28]
})

bins = [0, 30, 50, 100]
labels = ["young", "middle", "old"]
df["age_group"] = pd.cut(df["age"], bins=bins, labels=labels,
 ↪  right=False)
```

24) Кумулятивная сумма:

```
df = pd.DataFrame({
    "day": ["Mon", "Tue", "Wed", "Thu", "Fri"],
    "sales": [100, 150, 120, 200, 130]
})
df["cumsum"] = df["sales"].cumsum()
# [100, 250, 370, 570, 700]
```

25) Вычисление корреляции между столбцами:

```
df = pd.DataFrame({
    "A": [1, 2, 3, 4, 5],
    "B": [2, 4, 6, 8, 10],
    "C": [5, 4, 3, 2, 1]
})
corr_matrix = df.corr()
#      A    B     C
# A   1.0  1.0  -1.0
# B   1.0  1.0  -1.0
# C  -1.0 -1.0   1.0
```

26) Сохранение DataFrame в CSV:

```
df = pd.DataFrame({
    "name": ["Alice", "Bob"],
    "age": [25, 30]
})
df.to_csv("output.csv", index=False)
```

10. Matplotlib: Визуализация данных

10.1. Введение в Matplotlib

Matplotlib — основная библиотека для визуализации данных в Python. Создана Джоном Хантером в 2003 году по образцу системы построения графиков MATLAB. Библиотека позволяет создавать статические, анимированные и интерактивные визуализации.

Основные преимущества:

- **Гибкость** — полный контроль над каждым элементом графика.
- **Разнообразие** — более 40 типов графиков.
- **Интеграция** — работает с NumPy, Pandas, Seaborn.
- **Экспорт** — сохранение в PNG, PDF, SVG, EPS.

10.2. Архитектура Matplotlib

Matplotlib имеет иерархическую структуру:

- **Figure** — контейнер верхнего уровня (холст).
- **Axes** — область построения графика (может быть несколько на Figure).
- **Axis** — оси X и Y с делениями и метками.
- **Artist** — все видимые элементы (линии, текст, маркеры).

1) Импорт библиотеки:

```
import matplotlib.pyplot as plt
import numpy as np
```

10.3. Два способа построения графиков

10.3.1 Pyplot API (MATLAB-style)

Простой способ для быстрых графиков. Функции работают с текущей фигурой и осями.

2) Базовый график с pyplot:

```
x = np.linspace(0, 10, 100)
y = np.sin(x)

plt.plot(x, y)
plt.title("Sinus")
plt.xlabel("x")
plt.ylabel("sin(x)")
plt.show()
```

10.3.2 Object-Oriented API

Более гибкий способ. Создаём объекты Figure и Axes явно.

3) Объектно-ориентированный подход:

```
fig, ax = plt.subplots()

x = np.linspace(0, 10, 100)
y = np.sin(x)

ax.plot(x, y)
ax.set_title("Sinus")
ax.set_xlabel("x")
ax.set_ylabel("sin(x)")
plt.show()
```

10.4. Линейные графики (Line Plot)

4) Простой линейный график:

```
x = [1, 2, 3, 4, 5]
y = [2, 4, 6, 8, 10]

plt.plot(x, y)
plt.show()
```

5) Несколько линий на одном графике:

```
x = np.linspace(0, 2 * np.pi, 100)
y1 = np.sin(x)
y2 = np.cos(x)

plt.plot(x, y1, label="sin(x)")
plt.plot(x, y2, label="cos(x)")
plt.legend()
plt.title("Sinus и Cosinus")
plt.show()
```

6) Настройка стиля линии:

```
x = np.linspace(0, 10, 50)
y = np.exp(-x/10) * np.sin(x)

plt.plot(
    x, y,
    color="red",          # цвет линии
    linewidth=2,          # толщина
    linestyle="--",       # стиль: -, --, :, -.
    marker="o",           # маркеры
    markersize=5,         # размер маркеров
    alpha=0.8             # прозрачность
)
plt.show()
```

10.5. Форматы и маркеры

Matplotlib поддерживает формат-строку для быстрой настройки.

7) Формат-строка:

```
x = np.arange(0, 10, 1)
y = x ** 2

plt.plot(x, y, "ro--")       # r=red, o=circle, --=dashed
plt.plot(x, y + 10, "bs-")   # b=blue, s=square, -=solid
plt.plot(x, y + 20, "g^:")   # g=green, ^=triangle, :=dotted
plt.show()
```

Доступные маркеры:

- o — круг, s — квадрат, ^ — треугольник вверх.
- v — треугольник вниз, d — ромб, * — звезда.
- + — плюс, x — крестик, . — точка.

10.6. Точечные диаграммы (Scatter Plot)

8) Базовый scatter plot:

```
x = np.random.rand(50)
y = np.random.rand(50)

plt.scatter(x, y)
plt.title("Scatter Plot")
plt.show()
```

9) Scatter с разным размером и цветом:

```
x = np.random.rand(100)
y = np.random.rand(100)
colors = np.random.rand(100)         # значения для цвета
sizes = 1000 * np.random.rand(100)   # размеры точек

plt.scatter(x, y, c=colors, s=sizes, alpha=0.5, cmap="viridis")
plt.colorbar()  # цветовая шкала
plt.show()
```

10.7. Столбчатые диаграммы (Bar Plot)

10) Вертикальная столбчатая диаграмма:

```
categories = ["A", "B", "C", "D", "E"]
values = [23, 45, 56, 78, 32]

plt.bar(categories, values, color="skyblue", edgecolor="black")
plt.title("Bar Chart")
plt.xlabel("Категории")
plt.ylabel("Значения")
plt.show()
```

11) Горизонтальная столбчатая диаграмма:

```
categories = ["Python", "Java", "C++", "JavaScript", "Go"]
popularity = [30, 20, 15, 25, 10]

plt.barh(categories, popularity, color="coral")
plt.xlabel("Популярность (%)")
plt.title("Programming Languages Popularity")
plt.show()
```

12) Сгруппированные столбцы:

```
categories = ["Q1", "Q2", "Q3", "Q4"]
sales_2023 = [100, 120, 140, 160]
sales_2024 = [110, 130, 150, 180]

x = np.arange(len(categories))
width = 0.35

fig, ax = plt.subplots()
ax.bar(x - width/2, sales_2023, width, label="2023")
ax.bar(x + width/2, sales_2024, width, label="2024")
ax.set_xticks(x)
ax.set_xticklabels(categories)
ax.legend()
ax.set_title("Sales by Quarter")
plt.show()
```

13) Гистограмма:

```
data = np.random.randn(1000)  # нормальное распределение

plt.hist(data, bins=30, color="steelblue", edgecolor="white")
plt.title("Histogram")
plt.xlabel("Значение")
plt.ylabel("Частота")
plt.show()
```

14) Круговая диаграмма:

```
labels = ["Python", "Java", "C++", "Other"]
sizes = [40, 25, 20, 15]
colors = ["#ff9999", "#66b3ff", "#99ff99", "#ffcc99"]

plt.pie(sizes, labels=labels, colors=colors, autopct="%1.1f%%")
plt.title("Language Distribution")
plt.show()
```

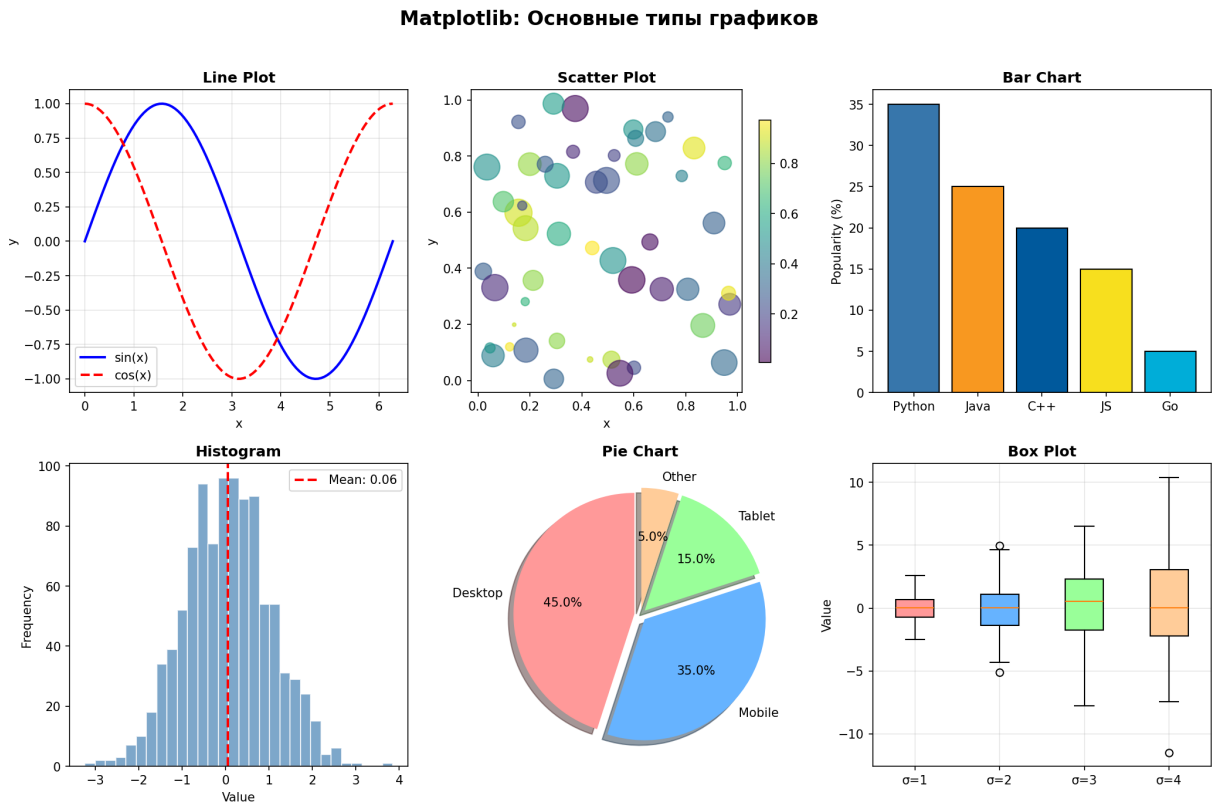


Рис. 16: Основные типы графиков: Line Plot, Scatter Plot, Bar Chart, Histogram, Pie Chart, Box Plot

10.8. Subplots (Несколько графиков)

15) Два графика рядом:

```
x = np.linspace(0, 10, 100)

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10, 4))

ax1.plot(x, np.sin(x), "b-")
ax1.set_title("Sinus")

ax2.plot(x, np.cos(x), "r-")
ax2.set_title("Cosinus")

plt.tight_layout()
plt.show()
```

10.9. Настройка графиков

16) Заголовки, метки, легенда:

```
x = np.linspace(0, 10, 100)
y1 = np.sin(x)
y2 = np.cos(x)

plt.plot(x, y1, label="sin(x)")
```

```
plt.plot(x, y2, label="cos(x)")

plt.title("Trigonometric Functions", fontsize=16,
 ↪  fontweight="bold")
plt.xlabel("X axis", fontsize=12)
plt.ylabel("Y axis", fontsize=12)
plt.legend(loc="upper right", fontsize=10)

plt.show()
```

17) Сетка и границы осей:

```
x = np.linspace(0, 10, 100)
y = np.sin(x)

plt.plot(x, y)
plt.xlim(0, 12)
plt.ylim(-1.5, 1.5)
plt.grid(True, linestyle="--", alpha=0.7)
plt.show()
```

18) Аннотации:

```
x = np.linspace(0, 10, 100)
y = np.sin(x)

plt.plot(x, y)
plt.annotate(
    "Maximum",
    xy=(np.pi/2, 1),
    xytext=(np.pi/2 + 1, 1.3),
    arrowprops=dict(facecolor="black", shrink=0.05),
    fontsize=12
)
plt.show()
```

10.10. Стили и сохранение

19) Применение стиля:

```
plt.style.use("ggplot")

x = np.linspace(0, 10, 100)
plt.plot(x, np.sin(x))
plt.plot(x, np.cos(x))
plt.title("ggplot Style")
plt.show()

plt.style.use("default")  # вернуть стиль по умолчанию
```

20) Сохранение в файл:

```
x = np.linspace(0, 10, 100)
y = np.sin(x)

plt.plot(x, y)
plt.title("Plot for Saving")

plt.savefig("my_plot.png", dpi=300, bbox_inches="tight")
plt.show()
```

10.11. Цвета и палитры, тепловые карты

21) Способы задания цвета:

```
x = np.linspace(0, 10, 100)

plt.plot(x, np.sin(x), color="red")           # по имени
plt.plot(x, np.sin(x+1), color="r")           # короткая форма
plt.plot(x, np.sin(x+2), color="#FF5733")     # HEX
plt.plot(x, np.sin(x+3), color=(0.1, 0.2, 0.5))   # RGB-кортеж
plt.show()
```

22) Базовая тепловая карта:

```
data = np.random.rand(8, 8)

plt.imshow(data, cmap="hot", interpolation="nearest")
plt.colorbar(label="Value")
plt.title("Heatmap")
plt.show()
```

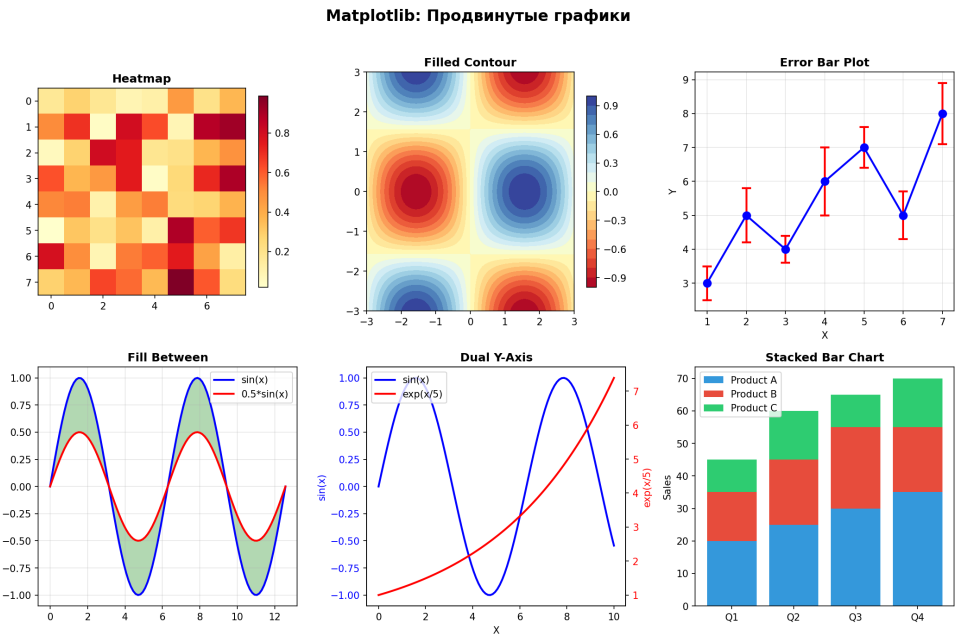


Рис. 17: Продвинутые графики: Heatmap, Filled Contour, Error Bar, Fill Between, Dual Y-Axis, Stacked Bar

10.12. Простые 3D-графики

23) 3D-кривая:

```
from mpl_toolkits.mplot3d import Axes3D  # noqa: F401

fig = plt.figure()
ax = fig.add_subplot(111, projection="3d")

t = np.linspace(0, 10*np.pi, 1000)
x = np.sin(t)
y = np.cos(t)
z = t

ax.plot(x, y, z)
ax.set_xlabel("X")
ax.set_ylabel("Y")
ax.set_zlabel("Z")
ax.set_title("3D Spiral")
plt.show()
```

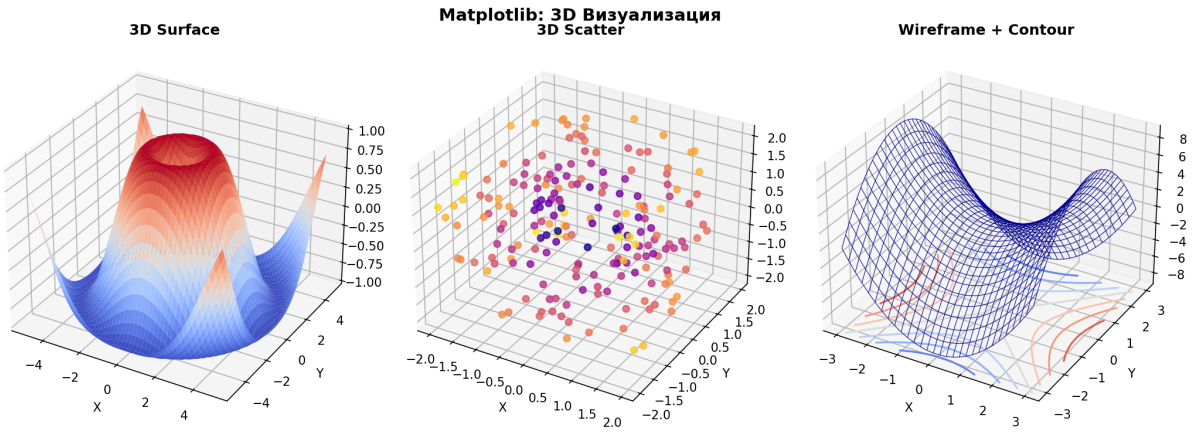


Рис. 18: 3D визуализация: Surface Plot, 3D Scatter, Wireframe.

11. Разбор заданий пробного тура МЭ

11.1. Задача A. Матрицы, которые коммутируют

11.1.1 Условие

Умножение матриц важно в машинном обучении: так описывают линейные слои нейросетей и последовательные преобразования признаков. Полезно помнить правило умножения. Если

$$M = \begin{pmatrix} p & q \\ r & s \end{pmatrix}, \quad N = \begin{pmatrix} u & v \\ w & x \end{pmatrix},$$

то произведение MN вычисляется по правилу «строка на столбец»:

$$MN = \begin{pmatrix} pu + qw & pv + qx \\ ru + sw & rv + sx \end{pmatrix}.$$

В общем случае перестановка множителей меняет результат: обычно $(MN \neq NM)$. Рассмотрим матрицы

$$A = \begin{pmatrix} 1 & a+2 \\ 1 & 1 \end{pmatrix}, \quad B = \begin{pmatrix} 3 & 7 \\ b+1 & 3 \end{pmatrix},$$

где a и b — целые числа. Известно, что произведения AB и BA совпадают:

$$AB = BA.$$

Найдите все целые значения a, для которых существует целое b, удовлетворяющее этому равенству.

11.1.2 Разбор

Запишем произведение AB:

$$AB = \begin{pmatrix} 1 & a+2 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} 3 & 7 \\ b+1 & 3 \end{pmatrix} = \begin{pmatrix} 3 + (a+2)(b+1) & 7 + 3(a+2) \\ 3 + (b+1) & 7 + 3 \end{pmatrix}$$

Запишем произведение BA:

$$BA = \begin{pmatrix} 3 & 7 \\ b+1 & 3 \end{pmatrix} \begin{pmatrix} 1 & a+2 \\ 1 & 1 \end{pmatrix} = \begin{pmatrix} 10 & 3(a+2) + 7 \\ (b+1) + 3 & (b+1)(a+2) + 3 \end{pmatrix}$$

Приравняем соответствующие элементы матриц ($AB = BA$):

1. Элемент (1,1): $3 + (a+2)(b+1) = 10 \implies (a+2)(b+1) = 7$.
2. Элемент (1,2): $7 + 3(a+2) = 3(a+2) + 7$ (верно всегда).
3. Элемент (2,1): $3 + (b+1) = (b+1) + 3$ (верно всегда).
4. Элемент (2,2): $7 + 3 = (b+1)(a+2) + 3 \implies (a+2)(b+1) = 7$.

Таким образом, единственное условие: произведение двух целых чисел $(a+2)$ и $(b+1)$ должно быть равно 7. Поскольку 7 — простое число, его делители: $1, 7, -1, -7$. Рассмотрим 4 случая для множителя $(a+2)$:

1. $a+2=1 \implies a=-1$ (тогда $b+1=7, b=6$)
2. $a+2=7 \implies a=5$ (тогда $b+1=1, b=0$)
3. $a+2=-1 \implies a=-3$ (тогда $b+1=-7, b=-8$)
4. $a+2=-7 \implies a=-9$ (тогда $b+1=-1, b=-2$)

Все найденные значения a целые, для каждого существует целое b. Упорядочим a по возрастанию.

Ответ: -9,-3,-1,5

11.2. Задача B. Максимум Энтропии

11.2.1 Условие

Во многих задачах по искусственному интеллекту важно оценивать, насколько модель уверена в своём ответе. Для этого смотрят на распределение вероятностей по вариантам: если один вариант почти наверняка верен, распределение сосредоточено, если же модель сильно сомневается, вероятности разных исходов ближе друг к другу. Такие числовые меры размазности распределения часто называют мерами энтропии и используют при обучении и настройке моделей.

Пусть задано дискретное распределение вероятностей $\{p_1, \dots, p_n\}$, $\sum_{i=1}^{n} p_i = 1$. Известно, что для всех i выполняется $p_i \geq 0{,}01$, и существует по крайней мере один исход j с вероятностью $p_j = 0{,}2$.

Найдите максимально возможное значение величины

$$H(p) = 1 - \sum_{i=1}^{n} p_i^2.$$

11.2.2 Разбор

Максимизация $H(p) = 1 - \sum p_i^2$ равносильна минимизации суммы квадратов $\sum p_i^2$. Сумма квадратов минимальна, когда слагаемые максимально близки друг к другу (распределение "равномерное"). Однако у нас есть ограничения.

Зафиксируем одно значение $p_1 = 0{,}2$. Оставшаяся сумма вероятностей $S = 1 - 0{,}2 = 0{,}8$. Эту сумму $0{,}8$ нужно распределить между оставшимися k элементами. Чтобы минимизировать сумму квадратов $\sum p_i^2$, нам выгодно, чтобы эти элементы были как можно меньше (квадрат малого числа значительно меньше квадрата большого). Минимально возможное значение $p_i = 0{,}01$. Посмотрим, сколько элементов по $0{,}01$ мы можем взять, чтобы покрыть сумму $0{,}8$:

$$k = \frac{0{,}8}{0{,}01} = 80$$

То есть мы можем взять ровно 80 элементов по 0.01. Проверим сумму: $0{,}2 + 80 \times 0{,}01 = 0{,}2 + 0{,}8 = 1$. Условие нормировки выполнено.

Считаем сумму квадратов для этого распределения:

$$\sum p_i^2 = (0{,}2)^2 + 80 \times (0{,}01)^2 = 0{,}04 + 80 \times 0{,}0001 = 0{,}04 + 0{,}008 = 0{,}048$$

Тогда энтропия:

$$H(p) = 1 - 0{,}048 = 0{,}952$$

Ответ: 0.952

11.3. Задача C. Проблемы с самооценкой

11.3.1 Условие

В классе учатся 11 человек, упорядоченных по уровню знаний: самый сильный — на 1-м месте, самый слабый — на 11-м. Петя находится ровно посередине — на 6-м месте.

В школе проводится контрольная работа. Каждый ученик может прийти на неё или пропустить. Чем выше уровень знаний ученика, тем лучше он напишет контрольную (строго лучше любого с более низким уровнем знаний). Также чем выше уровень знаний, тем больше вероятность, что ученик придёт.

Вероятность прихода ученика с местом k равна:

$$p_k = \frac{12 - k}{11}$$

Например, самый сильный (1-е место) приходит с вероятностью 1, а самый слабый (11-е место) — с вероятностью $\frac{1}{11}$.

Метрика самооценки Пети вычисляется так:

$$(\text{количество, кто написал хуже Пети}) - (\text{количество, кто написал лучше Пети})$$

Если бы на контрольную пришли все, метрика Пети была бы равна 0 (пятеро хуже, пятеро лучше).

Какое значение этой метрики в среднем будет у Пети, если ученики приходят независимо с указанными вероятностями?

11.3.2 Разбор

Метрика является линейной комбинацией случайных величин (индикаторов прихода каждого ученика). Введем индикатор I_k, равный 1, если ученик k пришел, и 0 иначе. $P(I_k = 1) = p_k$. Ученики приходят независимо. Петя (6-е место) делит остальных на две группы:

- Лучше Пети (места $1, 2, 3, 4, 5$).
- Хуже Пети (места $7, 8, 9, 10, 11$).

Обозначим N_{worse} и N_{better} количество пришедших из этих групп.

$$M = N_{worse} - N_{better}$$

$$E[M] = E[N_{worse}] - E[N_{better}]$$

Математическое ожидание суммы индикаторов равно сумме вероятностей:

$$E[N_{worse}] = \sum_{k=7}^{11} p_k = \sum_{k=7}^{11} \frac{12 - k}{11} = \frac{5 + 4 + 3 + 2 + 1}{11} = \frac{15}{11}$$

$$E[N_{better}] = \sum_{k=1}^{5} p_k = \sum_{k=1}^{5} \frac{12 - k}{11} = \frac{11 + 10 + 9 + 8 + 7}{11} = \frac{45}{11}$$

Итоговое ожидание:

$$E[M] = \frac{15}{11} - \frac{45}{11} = -\frac{30}{11} \approx -2{,}727273$$

Ответ: -2.72727

11.4. Задача D. Дерево решений

11.4.1 Условие

Вам дан файл data с данными о школьниках.
Формат данных:

- age — возраст (целое число от 10 до 18),
- math — балл по математике (целое число от 0 до 100),
- club — кружок (одно из четырёх значений: спорт, информатика, музыка, нет).

Каждая строка соответствует одному школьнику. Пропусков в данных нет.
Также дано дерево решений глубины до 3. Оно классифицирует каждого школьника в класс 0 или класс 1.
Сколько строк файла data будут отнесены этим деревом к классу 1?

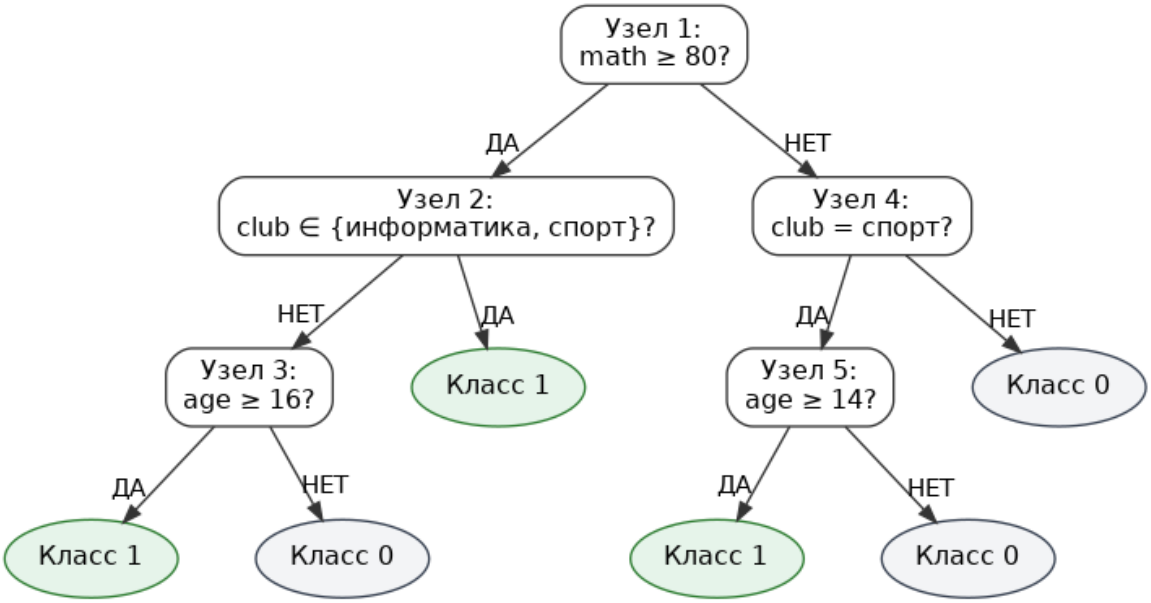


Рис. 19: Дерево решений для классификации школьников

11.4.2 Идейное решение

Задача сводится к программной реализации логики дерева (набора условий if-else). Мы должны пройтись по каждой строке данных и проверить условия, спускаясь от корня к листьям. Если попадаем в лист "Класс 1 увеличиваем счетчик.

11.4.3 Решение (Python)

```python
import csv


def classify(row):
    math_score = int(row['math'])
    age = int(row['age'])
    club = row['club']

    if math_score >= 80:
        if club in ['информатика', 'спорт']:
            return 1
        else:
            if age >= 16:
                return 1
            else:
                return 0
    else:
        if club == 'спорт':
            if age >= 14:
                return 1
            else:
                return 0
        else:
            return 0


with open('data.csv', 'r', encoding='utf-8') as file:
    reader = csv.DictReader(file)
    data = list(reader)

count = 0
for row in data:
    count += classify(row)
print(count)
```

11.4.4 Решение (Excel)

В Excel это можно реализовать через вложенные формулы **ЕСЛИ**.

После применения формулы ко всем строкам нужно просто просуммировать столбец с единицами. Достаточно аккуратно написать условия о принадлежности классам.

Ответ: 281

11.5. Задача E. Фильтр скрытых мыслей

11.5.1 Условие

Иногда в тексте встречаются служебные фрагменты, заключённые между маркерами [[и]]. Требуется удалить из строки все такие фрагменты вместе с маркерами. Гарантируется, что маркеры корректны и пары [[и]] не пересекаются и не вложены.

Примеры:

Ввод	Вывод
Answer: [[think...]]42	Answer: 42
[[draft]]Done!	Done!
A[[x]]B[[y]]C	ABC

11.5.2 Идейное решение

Так как вложенности нет, задача решается линейным проходом по строке. Мы идем по символам и поддерживаем состояние: "печатать"или "не печатать". Встретив '[[', перестаем записывать символы в ответ и ищем закрывающий ']]'. После нахождения ']]' пропускаем его и возобновляем запись.

11.5.3 Решение

```python
s = input()
res = []
i = 0
n = len(s)

while i < n:
    if i + 1 < n and s[i] == '[' and s[i+1] == '[':
        i += 2
        while i + 1 < n:
            if s[i] == ']' and s[i+1] == ']':
                i += 2
                break
            i += 1
    else:
        res.append(s[i])
        i += 1

result_str = "".join(res)
if not result_str:
    print("-")
else:
    print(result_str)
```

11.6. Задача F. Фильтрация датасета

11.6.1 Условие

При подготовке датасетов для обучения языковых моделей иногда генерируют синтетические строки из маленького алфавита — чтобы проверить, как модель учится на простых паттернах. Предположим, что мы генерируем строки только из символов "a "b"и "c".

Однако обработчик данных запрещает появление подстроки "ab": она считается служебной меткой, и если она попадёт в обучающую выборку, модель начнёт на неё переобучаться. Поэтому из всех строк длины n нам нужны только те, в которых ни разу не встречается подстрока "ab".

Найдите, сколько таких строк можно оставить в датасете.

Примеры:

Ввод	Вывод
0	1
3	21

11.6.2 Идейное решение

Задачу можно решить методом динамического программирования. Будем строить строку символ за символом и подсчитывать количество допустимых строк длины i, оканчивающихся на конкретный символ. Состояние динамики: $dp[i][last\_char]$ — количество правильных строк длины i, оканчивающихся на символ $last\_char$. Где $0 \to' a', 1 \to' b', 2 \to' c'$.

11.6.3 Решение

```python
n = int(input())
dp = [[0, 0, 0] for _ in range(n + 2)]
dp[0] = [1, 0, 0]
dp[1] = [1, 1, 1]

for i in range(2, n + 1):
    dp[i][0] = dp[i - 1][0] + dp[i - 1][1] + dp[i - 1][2]
    dp[i][1] = dp[i - 1][1] + dp[i - 1][2]
    dp[i][2] = dp[i - 1][0] + dp[i - 1][1] + dp[i - 1][2]

print(sum(dp[n]))
```

Пояснение к переходам:

Пусть мы построили строку длины $i - 1$. Попробуем добавить i-й символ:

- Добавляем **'a'**: можно добавить к строке, оканчивающейся на любой символ ('a', 'b', 'c'). Ограничений нет.

$$dp[i][0] = dp[i-1][0] + dp[i-1][1] + dp[i-1][2]$$

- Добавляем **'b'**: можно добавить только если предыдущий символ **не** был 'a' (иначе получится "ab"). Значит, можно добавить к строкам, оканчивающимся на 'b' или 'c'.

$$dp[i][1] = dp[i-1][1] + dp[i-1][2]$$

- Добавляем **'c'**: можно добавить к любому символу.

$$dp[i][2] = dp[i-1][0] + dp[i-1][1] + dp[i-1][2]$$

База динамики: $dp[0] = (1, 0, 0)$ (пустая строка), $dp[1] = (1, 1, 1)$ (строки "a "b "c").
Для примера $n = 3$:

1. $n = 1 : (1, 1, 1)$, sum=3.
2. $n = 2$:
   - $a : 1 + 1 + 1 = 3$ ("aa "ba "ca")
   - $b : 1 + 1 = 2$ ("bb "cb") - "ab"отпал
   - $c : 1 + 1 + 1 = 3$ ("ac "bc "cc")

   sum=8.
3. $n = 3$:
   - $a : 3 + 2 + 3 = 8$
   - $b : 2 + 3 = 5$
   - $c : 3 + 2 + 3 = 8$

   sum=21.

Ответ: 21

12. Scikit-learn

12.1. Что это и зачем

Scikit-learn (sklearn) — это библиотека Python для машинного обучения, построенная на NumPy, SciPy и Matplotlib. Она предоставляет готовые реализации алгоритмов для классификации, регрессии, кластеризации и других задач.

Основная идея scikit-learn: **все алгоритмы работают по одному и тому же паттерну**.

12.2. Единый интерфейс: fit, predict, score

В scikit-learn каждый алгоритм (модель) — это объект, называемый **estimator**. Все estimators имеют три основных метода:

- `fit(X, y)` — обучить модель на данных. X — признаки (features), y — целевая переменная (target).
- `predict(X)` — предсказать y для новых данных X.
- `score(X, y)` — оценить качество модели на данных (вернуть число от 0 до 1).

Этот паттерн работает для **всех алгоритмов**:

```
from sklearn.linear_model import LinearRegression
model = LinearRegression()
model.fit(X_train, y_train)
y_pred = model.predict(X_test)
score = model.score(X_test, y_test)
```

То же самое для логистической регрессии, деревьев решений, SVM, K-means и т.д. Меняется только название класса, логика одинакова.

12.3. Линейная регрессия: интуиция

Задача регрессии: предсказать число y на основе признаков $x_1, x_2, \dots, x_n$.

Идея линейной регрессии: найти такую прямую (или гиперплоскость в многомерном случае), которая лучше всего описывает связь между x и y.

Модель выглядит как:

$$\hat{y} = w_0 + w_1 x_1 + w_2 x_2 + \dots + w_n x_n$$

где w_i — веса (коэффициенты), которые модель **находит из данных**. w_0 называют **свободным членом** (intercept).

Пример. Предсказываем цену квартиры по площади:

цена $= 500\,000 + 50\,000 \cdot$ площадь

Смысл: квартира 20 м^2 будет стоить примерно $500\,000 + 50\,000 \cdot 20 = 1\,500\,000$ рублей.

12.4. Как модель находит веса

Веса w выбирают так, чтобы минимизировать **ошибку** модели на обучающих данных. Самая частая ошибка — **среднеквадратичная ошибка** (MSE):

$$\text{MSE} = \frac{1}{m}\sum_{i=1}^{m}(y_i - \hat{y}_i)^2$$

где m — количество примеров, y_i — реальное значение, $\hat{y}_i$ — предсказание.

Линейная регрессия в sklearn **решает эту оптимизацию аналитически** (через матричные операции), а не итеративно. Это быстро и точно.

12.5. Разделение данных: train и test

Если обучить модель на данных и оценить её на тех же данных — результат будет завышен. Модель просто **запомнила** примеры.

Правильный способ:

1. Разделить данные на **обучающую выборку** (80%) и **тестовую** (20%).
2. Обучить модель только на обучающей выборке.
3. Оценить качество на тестовой выборке (которую модель не видела).

Почему это важно: если модель хорошо работает на тестовой выборке, значит она **обобщилась** — научилась предсказывать не только на примерах, на которых училась.

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)
```

Аргумент `random_state=42` нужен для воспроизводимости: если запустить код дважды, разделение будет одинаковым.

12.6. Нормализация признаков

Признаки часто имеют разные масштабы. Например:

- Возраст: 18, 25, 35, 42 (десятки)
- Доход: 50000, 100000, 150000 (сотни тысяч)

Если признаки не нормализовать, модель будет неправильно работать (будет переоценивать важность больших чисел).

12.6.1 StandardScaler

Формула:

$$x'_j = \frac{x_j - \mu_j}{\sigma_j}$$

где μ_j — среднее, σ_j — стандартное отклонение.
Результат: признаки имеют среднее 0 и дисперсию 1.

```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

12.6.2 MinMaxScaler

Формула:

$$x'_j = \frac{x_j - \min(x_j)}{\max(x_j) - \min(x_j)}$$

Результат: все признаки в диапазоне $[0, 1]$.

```
from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

Методы:

- `fit_transform()` — вычислить параметры на X и преобразовать X.
- `transform()` — просто применить уже вычисленные параметры.

Если использовать `fit_transform()` на тестовой выборке, информация из тестовых данных **протечет** в обучение. Это называют **утечкой данных** (data leakage).

12.7. Метрики качества

После обучения нужно оценить, насколько хорошо модель работает.
Для регрессии используют:

- **MSE (Mean Squared Error)** — средняя квадратичная ошибка. Чем меньше, тем лучше.
- **MAE (Mean Absolute Error)** — средняя абсолютная ошибка. Более устойчива к выбросам.
- **R^2 (коэффициент детерминации)** — доля объясненной дисперсии. Изменяется от 0 до 1. Чем больше, тем лучше.
- **RMSE (Root Mean Squared Error)** — корень из MSE. Измеряется в том же масштабе, что целевая переменная.

```python
from sklearn.metrics import mean_squared_error, r2_score
import numpy as np

mse = mean_squared_error(y_test, y_pred)
rmse = np.sqrt(mse)
r2 = r2_score(y_test, y_pred)
```

12.8. SGDRegressor: линейная регрессия через градиентный спуск

В sklearn есть класс `SGDRegressor`, который реализует линейную регрессию через стохастический градиентный спуск.

Когда его использовать? Когда данных очень много (миллионы примеров). Обычный `LinearRegression` требует хранить все данные в памяти, а SGD может обрабатывать данные порциями.

Параметры:

- `loss='squared_error'` — функция потерь (здесь: среднеквадратичная ошибка).
- `max_iter` — максимум итераций обучения.
- `learning_rate='constant'` — как менять step size. 'constant' значит постоянный.
- `eta0=0.01` — величина шага.

```python
from sklearn.linear_model import SGDRegressor

model = SGDRegressor(
    loss='squared_error',
    max_iter=1000,
    learning_rate='constant',
    eta0=0.01
)
model.fit(X_train_scaled, y_train)
y_pred = model.predict(X_test_scaled)
```

12.9. Pipeline: объединяем нормализацию и модель

Часто нужно сначала нормализовать данные, потом обучить модель. Можно это делать отдельно, но удобнее объединить в один **pipeline**:

```python
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LinearRegression

pipeline = Pipeline([
    ('scaler', StandardScaler()),
    ('model', LinearRegression())
])

pipeline.fit(X_train, y_train)
y_pred = pipeline.predict(X_test)
```

Pipeline автоматически:

1. Вычисляет параметры нормализации на `X_train`.
2. Применяет нормализацию к `X_train` и обучает модель.
3. При предсказании сначала нормализует `X_test`, потом предсказывает.

Это гарантирует, что не будет утечки данных.

12.10. Простой пример: от начала до конца

```python
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score

# 1. Данные (синтетические)
np.random.seed(42)
n_samples = 100
X = np.random.randn(n_samples, 3)
# Целевая переменная: линейная комбинация + шум
y = 2.5*X[:, 0] + 3.2*X[:, 1] - 1.5*X[:, 2] +
 ↪  np.random.randn(n_samples)*0.5

# 2. Разделение
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

# 3. Нормализация
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# 4. Обучение
model = LinearRegression()
model.fit(X_train_scaled, y_train)

# 5. Предсказание
y_pred = model.predict(X_test_scaled)

# 6. Оценка
mse = mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)

print(f"MSE: {mse:.4f}")
print(f"R²: {r2:.4f}")
print(f"Веса модели: {model.coef_}")
print(f"Свободный член: {model.intercept_:.4f}")
```

12.11. Регуляризация: Ridge и Lasso регрессии

Проблема переобучения

При обучении модели линейной регрессии мы минимизируем функцию потерь:

$$J(w) = \frac{1}{m}\sum_{i=1}^{m}(y_i - \hat{y}_i)^2$$

Однако если у нас много признаков или мало данных, модель может **переобучиться** — слишком хорошо подстроиться под обучающую выборку, но плохо работать на новых данных.

Признаки переобучения:

- Очень высокое качество на обучающей выборке
- Значительно худшее качество на тестовой выборке
- Очень большие по модулю веса $|w_j|$

Решение: использовать регуляризацию — добавить к функции потерь штраф за слишком большие веса.

12.11.1 Ridge регрессия (L2 регуляризация)

Определение 12.1. Ridge регрессия

Ridge регрессия добавляет к функции потерь штраф за **квадраты весов**:

$$J(w) = \frac{1}{m}\sum_{i=1}^{m}(y_i - \hat{y}_i)^2 + \lambda\sum_{j=1}^{n} w_j^2$$

где:

- λ — параметр регуляризации (гиперпараметр)
- $\sum_{j=1}^{n} w_j^2$ — L2 норма весов в квадрате

Эффект:

- **Уменьшает все веса** пропорционально их величине
- Веса никогда не становятся точно равными нулю
- Все признаки остаются в модели
- Чем больше λ, тем сильнее регуляризация

Когда использовать?

- Все признаки потенциально важны
- Нужно уменьшить влияние отдельных признаков
- Есть подозрение на переобучение

Реализация в sklearn:

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

# Разделение данных
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

# Ridge регрессия
model_ridge = Ridge(alpha=1.0)  # alpha = λ
model_ridge.fit(X_train, y_train)

# Оценка
y_pred = model_ridge.predict(X_test)
print(f"R² = {r2_score(y_test, y_pred):.4f}")
print(f"MSE = {mean_squared_error(y_test, y_pred):.2f}")
```

Параметры:

- `alpha` — параметр регуляризации λ. По умолчанию `alpha=1.0`
- Чем больше `alpha`, тем сильнее регуляризация

12.11.2 Lasso регрессия (L1 регуляризация)

Определение 12.2. Lasso регрессия

Lasso регрессия добавляет штраф за **модули весов**:

$$J(w) = \frac{1}{m}\sum_{i=1}^{m}(y_i - \hat{y}_i)^2 + \lambda\sum_{j=1}^{n}|w_j|$$

где $\sum_{j=1}^{n}|w_j|$ — L1 норма весов.

Эффект:

- **Зануляет некоторые веса** полностью
- Выполняет автоматический **feature selection**
- Оставляет только самые важные признаки
- Интерпретируемость модели улучшается

Когда использовать?

- Подозрение, что многие признаки лишние
- Нужен автоматический отбор признаков
- Важна интерпретируемость (мало признаков)
- Большое количество признаков

Реализация в sklearn:

```python
from sklearn.linear_model import Lasso

# Lasso регрессия
model_lasso = Lasso(alpha=0.1)  # alpha = λ
model_lasso.fit(X_train, y_train)

# Оценка
y_pred = model_lasso.predict(X_test)
print(f"R² = {r2_score(y_test, y_pred):.4f}")

# Проверка, сколько весов занулилось
n_zero = (model_lasso.coef_ == 0).sum()
print(f"Нулевых весов: {n_zero} из {len(model_lasso.coef_)}")
```

Сравнение Ridge и Lasso

Критерий	Ridge (L2)	Lasso (L1)
Штраф	$\lambda \sum w_j^2$	$\lambda \sum \lvert w_j \rvert$
Эффект на веса	Уменьшает все	Зануляет некоторые
Когда использовать	Все признаки важны	Много лишних признаков
Интерпретируемость	Средняя	Высокая

Таблица 1: Сравнение Ridge и Lasso регрессий

Выбор параметра λ (alpha)

- $\lambda = 0$: обычная линейная регрессия (нет регуляризации)
- **Малое** λ: слабая регуляризация, возможно переобучение
- **Оптимальное** λ: баланс между обучением и обобщением
- **Большое** λ: сильная регуляризация, возможно недообучение

Как подобрать?
Используйте кросс-валидацию:

```python
from sklearn.model_selection import cross_val_score
import numpy as np

# Тестируем разные значения alpha
alphas = [0.001, 0.01, 0.1, 1.0, 10.0, 100.0]
scores = []

for alpha in alphas:
    model = Ridge(alpha=alpha)
    cv_scores = cross_val_score(model, X_train, y_train,
                                cv=5, scoring='r2')
    scores.append(cv_scores.mean())
    print(f"alpha={alpha:6.3f}: R² = {cv_scores.mean():.4f}")

# Лучшее значение
best_alpha = alphas[np.argmax(scores)]
print(f"\nЛучшее alpha = {best_alpha}")
```

Или используйте автоматический подбор:

```python
from sklearn.linear_model import RidgeCV, LassoCV

# Автоматический подбор alpha для Ridge
model_ridge_cv = RidgeCV(alphas=[0.1, 1.0, 10.0], cv=5)
model_ridge_cv.fit(X_train, y_train)
print(f"Лучшее alpha (Ridge): {model_ridge_cv.alpha_}")

# Автоматический подбор alpha для Lasso
model_lasso_cv = LassoCV(alphas=[0.01, 0.1, 1.0], cv=5)
model_lasso_cv.fit(X_train, y_train)
print(f"Лучшее alpha (Lasso): {model_lasso_cv.alpha_}")
```

Практический пример

```python
import numpy as np
import pandas as pd
from sklearn.datasets import make_regression
from sklearn.linear_model import LinearRegression, Ridge, Lasso
from sklearn.model_selection import train_test_split
from sklearn.metrics import r2_score

# Создаем данные с шумом
X, y = make_regression(n_samples=100, n_features=20,
                       n_informative=5, noise=10,
                       random_state=42)

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

# Сравниваем три модели
models = {
    'Linear Regression': LinearRegression(),
    'Ridge (alpha=1.0)': Ridge(alpha=1.0),
    'Lasso (alpha=0.1)': Lasso(alpha=0.1)
}

print("Сравнение моделей:\n")
print(f"{'Модель':<25} | {'Train R²':>10} | {'Test R²':>10}")
print("-" * 50)

for name, model in models.items():
    model.fit(X_train, y_train)
    train_score = model.score(X_train, y_train)
    test_score = model.score(X_test, y_test)
    print(f"{name:<25} | {train_score:>10.4f} |
     ↪  {test_score:>10.4f}")

    # Для Lasso покажем количество нулевых весов
    if isinstance(model, Lasso):
        n_zero = (model.coef_ == 0).sum()
        print(f"  → Нулевых весов: {n_zero} из {len(model.coef_)}")
```

Интерпретация результатов:

- Если Train R $\gg$ Test R для Linear Regression — переобучение
- Ridge уменьшит разницу, улучшив Test R
- Lasso может показать, сколько признаков реально важны

12.12. Работа с категориальными данными

Проблема категориальных признаков

Многие алгоритмы машинного обучения работают только с **числовыми** данными. Но в реальных задачах часто встречаются категориальные признаки:

- Город: “Москва”, “Санкт-Петербург”, “Казань”
- Цвет: “красный”, “зелёный”, “синий”
- Размер: “S”, “M”, “L”, “XL”
- Состояние: “новое”, “хорошее”, “среднее”, “плохое”

Нельзя просто присвоить номера! Модель может решить, что “Казань” = 3 > “Москва” = 1, хотя это не так.

12.12.1 OneHotEncoder

OneHotEncoder создаёт отдельный бинарный признак для каждой категории.
Пример:

$$\text{Город} = \begin{cases} \text{“Москва”} \to [1, 0, 0] \\ \text{“СПб”} \to [0, 1, 0] \\ \text{“Казань”} \to [0, 0, 1] \end{cases}$$

Когда использовать

- **Номинальные категории** — нет естественного порядка
- Примеры: город, цвет, тип товара, марка автомобиля

```python
import pandas as pd
from sklearn.preprocessing import OneHotEncoder

# Данные
data = pd.DataFrame({
    'city': ['Moskva', 'SPb', 'Kazan', 'Moskva', 'SPb']
})

# OneHotEncoder
encoder = OneHotEncoder(sparse_output=False)
encoded = encoder.fit_transform(data[['city']])

print("Исходные данные:")
print(data['city'].values)
print("\nЗакодированные данные:")
print(encoded)
print("\nКатегории:")
print(encoder.categories_)
```

Важные параметры:

- `sparse_output=False` — вернуть плотный массив (не разрежённый)
- `drop='first'` — удалить первую категорию (избежать мультиколлинеарности)
- `handle_unknown='ignore'` — игнорировать неизвестные категории

Проблема мультиколлинеарности
Если есть 3 категории с кодировками $[1, 0, 0]$, $[0, 1, 0]$, $[0, 0, 1]$, то третья колонка избыточна:

$$\text{col}_3 = 1 - \text{col}_1 - \text{col}_2$$

Решение: использовать `drop='first'`:

```
# Избегаем мультиколлинеарности
encoder = OneHotEncoder(drop='first', sparse_output=False)
encoded = encoder.fit_transform(data[['city']])

print("Закодировано с drop='first':")
print(encoded)  # Теперь только 2 колонки вместо 3
```

12.12.2 OrdinalEncoder

OrdinalEncoder присваивает числа в соответствии с **естественным порядком** категорий.
Пример:

$$\text{Размер} = \begin{cases} \text{“XS”} \to 0 \\ \text{“S”} \to 1 \\ \text{“M”} \to 2 \\ \text{“L”} \to 3 \\ \text{“XL”} \to 4 \end{cases}$$

Здесь порядок **важен**: XS < S < M < L < XL.
Когда использовать?

- **Порядковые категории** — есть естественный порядок
- Примеры: размер одежды, оценки (плохо/средне/хорошо), уровень образования

```
from sklearn.preprocessing import OrdinalEncoder

# Данные с порядком
data = pd.DataFrame({
    'size': ['M', 'L', 'S', 'XL', 'M', 'S']
})

# Явно задаем порядок
encoder = OrdinalEncoder(
    categories=[['XS', 'S', 'M', 'L', 'XL']]
)
encoded = encoder.fit_transform(data[['size']])

print("Исходные данные:")
print(data['size'].values)
print("\nЗакодированные данные:")
print(encoded.flatten())
```

Важно: всегда указывайте параметр `categories` явно, чтобы контролировать порядок!

12.12.3 LabelEncoder

LabelEncoder просто присваивает номера категориям:

$$\text{“красный”} \to 0, \quad \text{“зелёный”} \to 1, \quad \text{“синий”} \to 2$$

Когда использовать?

- **Только для целевой переменной** y в задачах классификации
- **НЕ использовать для признаков!** (создаст искусственный порядок)

Реализация в sklearn

```python
from sklearn.preprocessing import LabelEncoder

# Целевая переменная
y = ['cat', 'dog', 'bird', 'cat', 'dog']

# LabelEncoder
encoder = LabelEncoder()
y_encoded = encoder.fit_transform(y)

print("Исходные метки:")
print(y)
print("\nЗакодированные метки:")
print(y_encoded)
print("\nКлассы:")
print(encoder.classes_)

# Обратное преобразование
y_decoded = encoder.inverse_transform(y_encoded)
print("\nДекодирование:")
print(y_decoded)
```

Сравнение энкодеров

Энкодер	Тип категорий	Результат	Для чего
OneHotEncoder	Номинальные	Несколько колонок 0/1	Признаки
OrdinalEncoder	Порядковые	Одна колонка (числа)	Признаки
LabelEncoder	Любые	Одна колонка (числа)	Целевая переменная

Таблица 2: Сравнение энкодеров категориальных данных

Правило выбора:

- Нет порядка (город, цвет) → **OneHotEncoder**
- Есть порядок (размер, оценка) → **OrdinalEncoder**
- Целевая переменная → **LabelEncoder**

12.13. ColumnTransformer

В реальных задачах датасеты содержат **разные типы признаков**:

- Числовые (площадь, возраст, цена)
- Категориальные (город, цвет, тип)

Нужно применить **разные трансформации** к разным группам признаков:

- Числовые → StandardScaler (нормализация)
- Категориальные → OneHotEncoder (кодирование)

Решение: ColumnTransformer
ColumnTransformer позволяет применять разные трансформации к разным колонкам.

```
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import StandardScaler, OneHotEncoder

# Определяем типы признаков
numeric_features = ['area', 'rooms', 'age']
categorical_features = ['city', 'condition']

# Создаем трансформер
preprocessor = ColumnTransformer(
    transformers=[
        ('num', StandardScaler(), numeric_features),
        ('cat', OneHotEncoder(drop='first'), categorical_features)
    ])
```

Параметры:

- `transformers` — список кортежей `(имя, трансформер, колонки)`
- `'num'`, `'cat'` — имена шагов (любые строки)
- Можно использовать индексы колонок или имена

Пример использования

```
import pandas as pd
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.pipeline import Pipeline
from sklearn.linear_model import Ridge
from sklearn.model_selection import train_test_split

# Создаем данные
data = pd.DataFrame({
    'area': [50, 70, 100, 120, 80],
    'rooms': [2, 3, 4, 5, 3],
    'age': [10, 5, 20, 15, 8],
    'city': ['Moskva', 'SPb', 'Moskva', 'Kazan', 'SPb'],
    'condition': ['Good', 'Excellent', 'Fair', 'Good',
     ↪  'Excellent'],
```

```
    'price': [5000000, 6000000, 8000000, 9000000, 6500000]
})

# Разделяем признаки и целевую переменную
X = data.drop('price', axis=1)
y = data['price']

# Определяем типы признаков
numeric_features = ['area', 'rooms', 'age']
categorical_features = ['city', 'condition']

# Создаем preprocessing pipeline
numeric_transformer = Pipeline(steps=[
    ('scaler', StandardScaler())
])

categorical_transformer = Pipeline(steps=[
    ('onehot', OneHotEncoder(drop='first', sparse_output=False))
])

preprocessor = ColumnTransformer(
    transformers=[
        ('num', numeric_transformer, numeric_features),
        ('cat', categorical_transformer, categorical_features)
    ])

# Полный pipeline с моделью
full_pipeline = Pipeline(steps=[
    ('preprocessor', preprocessor),
    ('regressor', Ridge(alpha=100.0))
])

# Разделяем данные
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

# Обучаем
full_pipeline.fit(X_train, y_train)

# Оцениваем
score = full_pipeline.score(X_test, y_test)
print(f"R² на тестовой выборке: {score:.4f}")

# Предсказываем
y_pred = full_pipeline.predict(X_test)
print("\nПримеры предсказаний:")
for true_val, pred_val in zip(y_test, y_pred):
    print(f"Истина: {true_val:,.0f}, Предсказание: {pred_val:,.0f}")
```

Преимущества ColumnTransformer:

1. **Автоматизация:** не нужно вручную обрабатывать каждую колонку

2. **Избегание утечки данных:** все трансформации обучаются только на train
3. **Удобство:** один объект для всех преобразований
4. **Воспроизводимость:** можно сохранить и использовать позже

Интеграция с Pipeline
ColumnTransformer отлично работает с Pipeline:

```
# Полный workflow
from sklearn.linear_model import Lasso

full_model = Pipeline([
    ('preprocessor', preprocessor),   # ColumnTransformer
    ('model', Lasso(alpha=1.0))       # Любая модель
])

# Обучение одной строкой
full_model.fit(X_train, y_train)

# Предсказание одной строкой
y_pred = full_model.predict(X_test)

# Оценка одной строкой
score = full_model.score(X_test, y_test)
```

12.14. GridSearchCV: автоматический подбор гиперпараметров

Проблема подбора гиперпараметров
До сих пор мы вручную выбирали параметры моделей:

- Ridge: $\alpha = 1.0$
- Lasso: $\alpha = 0.1$
- SGDRegressor: `learning_rate`, `max_iter`, `eta0`

Но как узнать, какие значения **оптимальны**? Перебирать вручную долго и неэффективно.
Решение: использовать `GridSearchCV` — автоматический подбор гиперпараметров.
Параметры vs Гиперпараметры
Важно различать:

- **Параметры модели** — веса $w_0, w_1, \ldots, w_n$, которые модель **находит сама** при обучении (через `fit`)
- **Гиперпараметры** — настройки модели (`alpha`, `max_iter`, `penalty`), которые мы **задаём ДО обучения**

Пример:

```
model = Ridge(alpha=1.0)  # alpha — гиперпараметр
model.fit(X_train, y_train)  # модель находит параметры (веса)
print(model.coef_)  # параметры модели
```

Что делает GridSearchCV

`GridSearchCV` автоматически:

1. Перебирает все комбинации гиперпараметров из заданной **сетки** (grid)
2. Для каждой комбинации проводит **кросс-валидацию** (обычно 5-fold)
3. Выбирает лучшую комбинацию по метрике качества
4. Обучает финальную модель на всех обучающих данных с лучшими параметрами

Преимущества:

- Автоматизация: не нужно перебирать вручную
- Кросс-валидация: защита от переобучения
- Универсальность: работает с любой моделью sklearn
- Параллелизм: можно ускорить с помощью `n_jobs=-1`

Простой пример: подбор alpha для Ridge

```python
from sklearn.model_selection import GridSearchCV
from sklearn.linear_model import Ridge

# Определяем сетку гиперпараметров
param_grid = {
    'alpha': [0.001, 0.01, 0.1, 1.0, 10.0, 100.0, 1000.0]
}

# Создаем GridSearchCV
grid_search = GridSearchCV(
    estimator=Ridge(),              # Модель
    param_grid=param_grid,          # Сетка параметров
    cv=5,                           # 5-fold кросс-валидация
    scoring='r2',                   # Метрика для оптимизации
    n_jobs=-1,                      # Использовать все ядра CPU
    verbose=1                       # Показывать прогресс
)

# Обучаем (перебираем все комбинации)
grid_search.fit(X_train, y_train)

# Результаты
print(f"Лучшие параметры: {grid_search.best_params_}")
print(f"Лучший CV score: {grid_search.best_score_:.4f}")
print(f"Test R²: {grid_search.score(X_test, y_test):.4f}")

# Лучшая модель
best_model = grid_search.best_estimator_
```

Что произошло?

- GridSearchCV перебрал 7 значений alpha

- Для каждого провёл 5-fold CV
- Итого: $7 \times 5 = 35$ обучений модели
- Выбрал alpha с лучшим средним CV score

Результаты всех комбинаций
После обучения можно посмотреть детальные результаты:

```python
import pandas as pd

# Таблица всех результатов
results = pd.DataFrame(grid_search.cv_results_)
summary = results[['param_alpha', 'mean_test_score',
                   'std_test_score', 'rank_test_score']]
summary = summary.sort_values('rank_test_score')

print(summary)
```

Результат:

```
  param_alpha  mean_test_score  std_test_score  rank_test_score
4        10.0           0.8523          0.0312                1
3         1.0           0.8501          0.0305                2
5       100.0           0.8412          0.0289                3
...
```

Расширенный пример: множество параметров
Можно подбирать несколько гиперпараметров одновременно:

```python
from sklearn.linear_model import Lasso

# Сетка с несколькими параметрами
param_grid = {
    'alpha': [0.001, 0.01, 0.1, 1.0, 10.0],
    'max_iter': [1000, 5000, 10000],
    'tol': [1e-4, 1e-3]
}

# Количество комбинаций: 5 × 3 × 2 = 30
# С 5-fold CV: 30 × 5 = 150 обучений!

grid_lasso = GridSearchCV(
    estimator=Lasso(),
    param_grid=param_grid,
    cv=5,
    scoring='r2',
    n_jobs=-1
)

grid_lasso.fit(X_train, y_train)

print(f"Лучшие параметры:")
for param, value in grid_lasso.best_params_.items():
```

```
    print(f"  {param}: {value}")

print(f"\nЛучший CV score: {grid_lasso.best_score_:.4f}")
print(f"Test R²: {grid_lasso.score(X_test, y_test):.4f}")
```

GridSearchCV с Pipeline

GridSearchCV отлично работает с Pipeline. Для параметров используют формат `'название_ша`

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler

# Pipeline: нормализация + модель
pipeline = Pipeline([
    ('scaler', StandardScaler()),
    ('model', Ridge())
])

# Сетка параметров для pipeline
param_grid = {
    'model__alpha': [0.1, 1.0, 10.0, 100.0],  # параметр модели
    'scaler': [StandardScaler(), MinMaxScaler(), None]  # тип
    ↪ нормализации
}

grid_pipeline = GridSearchCV(
    estimator=pipeline,
    param_grid=param_grid,
    cv=5,
    scoring='r2',
    n_jobs=-1
)

grid_pipeline.fit(X_train, y_train)

print(f"Лучшие параметры:")
print(f"  Alpha: {grid_pipeline.best_params_['model__alpha']}")
scaler_type = grid_pipeline.best_params_['scaler']
scaler_name = type(scaler_type).__name__ if scaler_type else 'None'
print(f"  Scaler: {scaler_name}")
```

Результат: GridSearchCV подобрал и гиперпараметры модели, и тип нормализации!

Основные параметры GridSearchCV

Полезные атрибуты после fit

После обучения GridSearchCV предоставляет:

- `best_params_` — лучшие параметры (словарь)
- `best_score_` — лучший CV score (число)
- `best_estimator_` — модель с лучшими параметрами (объект)
- `cv_results_` — детальные результаты всех комбинаций (словарь)

Параметр	Описание
`estimator`	Модель (любой sklearn estimator)
`param_grid`	Словарь с сеткой параметров
`cv`	Количество фолдов для кросс-валидации (по умолчанию 5)
`scoring`	Метрика для оптимизации ('r2', 'neg_mse', 'accuracy'...)
`n_jobs`	Количество ядер CPU (-1 = все доступные)
`verbose`	Уровень детализации вывода (0=тихо, 1=прогресс, 2=подробно)
`refit`	Переобучить на всех данных с лучшими параметрами (по умолчанию True)

Таблица 3: Основные параметры GridSearchCV

Пример использования:

```
# Получаем лучшую модель
best_model = grid_search.best_estimator_

# Предсказываем напрямую через GridSearchCV
y_pred = grid_search.predict(X_test)

# Или используем лучшую модель
y_pred = best_model.predict(X_test)
```

Выбор метрики scoring
Для регрессии:

- `'r2'` — R score (по умолчанию)
- `'neg_mean_squared_error'` — отрицательный MSE
- `'neg_mean_absolute_error'` — отрицательный MAE
- `'neg_root_mean_squared_error'` — отрицательный RMSE

Для классификации:

- `'accuracy'` — accuracy
- `'precision'`, `'recall'`, `'f1'` — соответствующие метрики
- `'roc_auc'` — площадь под ROC-кривой

Почему отрицательные? В sklearn scoring функции **максимизируются**, а MSE нужно минимизировать. Поэтому используют отрицательные значения.

Как составить сетку параметров?
Стратегия грубого-точного поиска:

1. **Грубый поиск:** широкий диапазон, мало точек

```
param_grid_coarse = {
    'alpha': [0.001, 0.1, 10, 1000]  # логарифмическая шкала
}
```

2. **Точный поиск:** узкий диапазон вокруг лучшего значения, больше точек

```
# Допустим, лучшее alpha = 10
param_grid_fine = {
    'alpha': [5, 7, 10, 15, 20]  # уточняем вокруг 10
}
```

Для alpha используйте логарифмическую шкалу:

```
import numpy as np
alphas = np.logspace(-3, 3, 7)  # [0.001, 0.01, 0.1, 1, 10, 100,
 ↪  1000]
```

Сравнение: ручной подбор vs GridSearchCV
Ручной подбор:

```
best_score = -np.inf
best_alpha = None

for alpha in [0.001, 0.01, 0.1, 1.0, 10.0, 100.0]:
    model = Ridge(alpha=alpha)
    scores = cross_val_score(model, X_train, y_train, cv=5,
     ↪  scoring='r2')
    mean_score = scores.mean()
    if mean_score > best_score:
        best_score = mean_score
        best_alpha = alpha

print(f"Лучший alpha: {best_alpha}")
print(f"Лучший CV score: {best_score:.4f}")
```

GridSearchCV:

```
param_grid = {'alpha': [0.001, 0.01, 0.1, 1.0, 10.0, 100.0]}
grid = GridSearchCV(Ridge(), param_grid, cv=5, scoring='r2',
 ↪  n_jobs=-1)
grid.fit(X_train, y_train)

print(f"Лучший alpha: {grid.best_params_['alpha']}")
print(f"Лучший CV score: {grid.best_score_:.4f}")
```

Результат одинаковый, но GridSearchCV:

- Короче и понятнее
- Быстрее (благодаря параллелизму)
- Автоматически сохраняет лучшую модель
- Предоставляет детальные результаты

Практические советы
Осторожно с размером сетки:

- Вычислительная сложность: количество обучений = (размер сетки) × (cv фолдов)

- Пример: сетка $5 \times 3 \times 2 = 30$ комбинаций, 5-fold CV $30 \times 5 = 150$ обучений
- Для больших датасетов или моделей это может занять часы

Переобучение на валидации:

- Если перебирать слишком много параметров, можно "переобучиться" на валидационной выборке
- Решение: использовать отдельную тестовую выборку для финальной оценки

12.15. Итоги

Основные концепции:

- Все алгоритмы в sklearn следуют паттерну: `fit`, `predict`, `score`.
- Данные всегда разделяют на обучающую и тестовую выборки.
- Нормализация критична для градиентных методов.
- Метрики качества показывают, насколько хорошо модель работает.
- Pipeline объединяет несколько шагов обработки в один объект.

13. Логистическая регрессия

13.1. От линейной регрессии к классификации

Задача регрессии: предсказать число (цена, вес, возраст).
Задача классификации: предсказать класс (спам/не спам, болезнь/здоров, кот/собака).
Если в задаче только два класса (например, 0 и 1), это называется **бинарная классификация**.
Почему нельзя просто применить линейную регрессию? Попытаемся.
Пусть предсказываем: будет ли клиент брать кредит (1) или нет (0).
Признак: доход клиента.
Если обучить линейную регрессию, модель выдаст:

$$\hat{y} = w_0 + w_1 \cdot \text{доход}$$

Проблемы:

1. Модель может вернуть $\hat{y} = -0.5$ или $\hat{y} = 3.7$, а это не класс.
2. Нужно предсказывать **вероятность** класса, а не само значение.

Решение: применить специальную функцию, которая приводит линейный выход в диапазон [0, 1].

13.2. Сигмоидная функция

Сигмоида — это функция, которая преобразует любое число в диапазон [0, 1]:

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

где $z = w_0 + w_1 x_1 + w_2 x_2 + ... + w_n x_n$ — линейный выход.
Интуиция:

- Если z очень большое (например, $z = 100$), то $e^{-z} \approx 0$, и $\sigma(z) \approx 1$.
- Если z очень маленькое (например, $z = -100$), то e^{-z} очень большое, и $\sigma(z) \approx 0$.
- Если $z = 0$, то $\sigma(z) = \frac{1}{1+1} = 0.5$.

Сигмоида имеет форму буквы S. Её основное свойство: $\sigma'(z) = \sigma(z)(1 - \sigma(z))$ — производная выражается через саму сигмоиду.

13.3. Модель логистической регрессии

В логистической регрессии предсказание вероятности класса 1:

$$P(y = 1|x) = \sigma(w_0 + w_1 x_1 + ... + w_n x_n) = \frac{1}{1 + e^{-(w_0 + w_1 x_1 + ... + w_n x_n)}}$$

Вероятность класса 0:

$$P(y = 0|x) = 1 - P(y = 1|x)$$

Предсказание:

- Если $P(y = 1|x) \geq 0.5$, предсказываем класс 1.
- Если $P(y = 1|x) < 0.5$, предсказываем класс 0.

Порог 0.5 можно менять в зависимости от задачи (подробнее ниже).

13.4. Интерпретация

Давайте разберем, что выдает сигмоида.

Пример. Предсказываем готовность клиента брать кредит. Модель выдала:

- $P(y = 1|x) = 0.9$ — с вероятностью 90% клиент возьмет кредит.
- $P(y = 1|x) = 0.3$ — с вероятностью 30% клиент возьмет кредит (или 70% не возьмет).
- $P(y = 1|x) = 0.5$ — модель полностью не уверена.

Эти вероятности очень полезны:

- Можно выставить разные пороги для разных сценариев.
- Можно оценить **уверенность** модели в своем предсказании.
- Можно ранжировать примеры по вероятности.

13.5. Функция потерь: cross-entropy

Теперь нужно понять, как найти веса $w_0, w_1, \dots, w_n$.

В линейной регрессии минимизировали MSE. Здесь нужна другая функция потерь, которая работает с вероятностями.

Cross-entropy loss (логистическая функция потерь):

$$L(y, \hat{y}) = -[y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})]$$

где $\hat{y} = \sigma(z)$ — предсказанная вероятность, $y \in \{0, 1\}$ — истинный класс.

Интуиция:

- Если $y = 1$ (истинный класс), функция становится $L = -\log(\hat{y})$.

  Если модель предсказала $\hat{y} = 0.9$ (близко к 1), то $\log(0.9) \approx -0.105$, ошибка мала.

  Если модель предсказала $\hat{y} = 0.1$ (далеко от 1), то $\log(0.1) \approx -2.3$, ошибка большая.
- Если $y = 0$ (истинный класс), функция становится $L = -\log(1 - \hat{y})$.

  Если модель предсказала $\hat{y} = 0.1$ (близко к 0), то $\log(0.9) \approx -0.105$, ошибка мала.

  Если модель предсказала $\hat{y} = 0.9$ (далеко от 0), то $\log(0.1) \approx -2.3$, ошибка большая.

На целой выборке:

$$J(w) = \frac{1}{m} \sum_{i=1}^{m} L(y_i, \hat{y}_i) = \frac{1}{m} \sum_{i=1}^{m} [-y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)]$$

где m — количество примеров.

Эту функцию называют **binary cross-entropy** или **log loss**.

13.6. Почему именно cross-entropy?

Cross-entropy возникает из **вероятностного подхода** и метода максимального правдоподобия.

Представим, что каждый пример i был сгенерирован независимо из распределения Бернулли с вероятностью $p_i = P(y = 1|x_i)$.

Вероятность всех наблюдений (правдоподобие):

$$L(w) = \prod_{i=1}^{m} p_i^{y_i}(1 - p_i)^{1-y_i}$$

Логарифм правдоподобия:

$$\log L(w) = \sum_{i=1}^{m}[y_i \log(p_i) + (1 - y_i) \log(1 - p_i)]$$

Максимизация правдоподобия эквивалентна минимизации:

$$-\frac{1}{m} \log L(w) = \frac{1}{m} \sum_{i=1}^{m}[-y_i \log(p_i) - (1 - y_i) \log(1 - p_i)]$$

Это в точности наша cross-entropy функция потерь!

Вывод: cross-entropy имеет глубокий вероятностный смысл. Это не просто выбор функции потерь, это логически следует из предположения, что данные порождены распределением Бернулли.

13.7. Нахождение весов

Минимизировать cross-entropy аналитически нельзя (в отличие от линейной регрессии). Используют **градиентный спуск**.

На каждой итерации вычисляют градиент функции потерь по весам:

$$\frac{\partial J}{\partial w_j} = \frac{1}{m} \sum_{i=1}^{m}(\hat{y}_i - y_i)x_{ij}$$

где x_{ij} — j-й признак примера i.

Это удивительно просто! Это (ошибка) $\times$ (признак).

Затем обновляют веса:

$$w_j := w_j - \eta \frac{\partial J}{\partial w_j}$$

где η — learning rate.

13.8. Регуляризация

Как и в линейной регрессии, добавляют штраф на размер весов, чтобы избежать переобучения.

L2-регуляризация (Ridge):

$$J(w) = \frac{1}{m} \sum_{i=1}^{m} L(y_i, \hat{y}_i) + \frac{\lambda}{2m} \sum_{j=1}^{n} w_j^2$$

L1-регуляризация (Lasso):

$$J(w) = \frac{1}{m}\sum_{i=1}^{m} L(y_i, \hat{y}_i) + \frac{\lambda}{m}\sum_{j=1}^{n} |w_j|$$

Параметр λ контролирует регуляризации:

- $\lambda = 0$ — нет регуляризации.
- λ большое — сильный штраф на веса, модель упрощается.

13.9. Матрицы ошибок и метрики

При классификации используют другие метрики, чем при регрессии.
Confusion matrix (матрица ошибок):

	Предсказано 0	Предсказано 1
Истинно 0	TN (true negative)	FP (false positive)
Истинно 1	FN (false negative)	TP (true positive)

На основе confusion matrix вычисляют:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

Доля правильных предсказаний.

$$\text{Precision} = \frac{TP}{TP + FP}$$

Из предсказанных положительных, сколько на самом деле положительных. *Когда важна мало ложных срабатываний: спам-фильтр.*

$$\text{Recall} = \frac{TP}{TP + FN}$$

Из истинных положительных, сколько модель нашла. *Когда важно не пропустить: поиск болезни.*

$$\text{F1} = 2 \cdot \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

Гармоническое среднее Precision и Recall.

13.10. ROC и AUC

Когда меняют порог классификации (не обязательно 0.5), меняются метрики.
ROC-кривая: график зависимости True Positive Rate (TPR) от False Positive Rate (FPR).

$$\text{TPR} = \frac{TP}{TP + FN} = \text{Recall}$$

$$\text{FPR} = \frac{FP}{FP + TN}$$

AUC (Area Under Curve): площадь под ROC-кривой.

- AUC = 1 — идеальная модель.
- AUC = 0.5 — случайный классификатор.
- AUC < 0.5 — модель хуже случайной (что-то не так).

AUC показывает: *если взять случайный положительный и случайный отрицательный пример, с какой вероятностью модель их правильно упорядочит.*

13.10.1 Precision-Recall кривая

Для несбалансированных данных **PR-кривая** часто информативнее ROC:

- **Ось X**: Recall
- **Ось Y**: Precision

Baseline для PR-AUC = доля положительного класса (например, 0.01 для 1% положительных).

13.11. Мультиклассовая классификация

Когда классов больше двух (не только 0 и 1), используют **softmax**.
Для каждого класса k вычисляют линейный выход $z_k = w_k \cdot x$.
Вероятность класса k:

$$P(y = k|x) = \frac{e^{z_k}}{\sum_{j=1}^{K} e^{z_j}}$$

Это обобщение сигмоиды на множество классов.
Функция потерь становится **categorical cross-entropy**:

$$L(y, \hat{y}) = -\sum_{k=1}^{K} y_k \log(\hat{y}_k)$$

где $y_k = 1$ если истинный класс k, и 0 иначе (one-hot encoding).

13.12. Как выбрать порог

Порог 0.5 работает только если классы сбалансированы и все ошибки одинаково дорогие.

На практике часто $P(y = 1|x) > 0.5$ меняют на $P(y = 1|x) > t$ для некоторого t.

Примеры:

- *Спам-фильтр:* ложный положительный (письмо в спаме) хуже, чем ложный отрицательный (спам в письмах). Повышаем порог (требуем $P > 0.7$).
- *Диагностика болезни:* пропустить болезнь хуже, чем ложная тревога. Понижаем порог (требуем $P > 0.3$).

Порог выбирают по ROC-кривой, смотря на точку, которая лучше всего подходит для задачи.

13.13. Итоги теории

- **Логистическая регрессия** — это линейная модель для классификации. На выход линейного предсказателя применяют сигмоиду, чтобы получить вероятность.
- **Сигмоида** преобразует любое число в диапазон (0, 1) и имеет простую производную.
- **Cross-entropy** — функция потерь, следует из вероятностного подхода (максимум правдоподобия)
- **Веса** находят градиентным спуском, градиент простой: (ошибка) $\times$ (признак).
- **Метрики** для классификации: Accuracy, Precision, Recall, F1, ROC, AUC.
- **Порог** классификации можно менять, зависит от задачи.
- **Softmax** обобщает сигмоиду на мультиклассовую классификацию.

13.14. Применение в scikit-learn

13.14.1 Простой пример

```python
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, confusion_matrix,
 ↪  classification_report

# Загружаем данные
iris = load_iris()
X = iris.data
y = iris.target

# Берем только два класса (0 и 1) для бинарной классификации
mask = (y == 0) | (y == 1)
X = X[mask]
y = y[mask]

# Разделяем данные
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

# Создаем и обучаем модель
model = LogisticRegression()
model.fit(X_train, y_train)

# Предсказываем
y_pred = model.predict(X_test)

# Оцениваем
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy:.4f}")
print(f"\nConfusion Matrix:\n{confusion_matrix(y_test, y_pred)}")
print(f"\nClassification Report:\n{classification_report(y_test,
 ↪  y_pred)}")
```

13.14.2 Работа с вероятностями

```python
# Получить вероятности вместо классов
y_proba = model.predict_proba(X_test)
print(f"Вероятности:\n{y_proba[:5]}")

# Вероятность класса 1
y_proba_class1 = model.predict_proba(X_test)[:, 1]
print(f"P(y=1): {y_proba_class1[:5]}")
```

13.14.3 Изменение порога

```python
from sklearn.metrics import roc_curve, auc

# ROC-кривая
fpr, tpr, thresholds = roc_curve(y_test, y_proba_class1)
roc_auc = auc(fpr, tpr)
print(f"AUC: {roc_auc:.4f}")

# Пользовательский порог
threshold = 0.6
y_pred_custom = (y_proba_class1 > threshold).astype(int)
accuracy_custom = accuracy_score(y_test, y_pred_custom)
print(f"Accuracy при пороге 0.6: {accuracy_custom:.4f}")
```

13.14.4 Мультиклассовая классификация

```python
# Используем все три класса
X = iris.data
y = iris.target

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

# LogisticRegression автоматически работает с несколькими классами
model = LogisticRegression(max_iter=200)
model.fit(X_train, y_train)

y_pred = model.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy (3 класса): {accuracy:.4f}")

# Вероятности для каждого класса
y_proba = model.predict_proba(X_test)
print(f"Форма y_proba: {y_proba.shape}")
print(f"Вероятности первого примера: {y_proba[0]}")
```

13.14.5 Регуляризация

```python
# L2-регуляризация (по умолчанию)
model_l2 = LogisticRegression(penalty='l2', C=1.0, max_iter=200)
model_l2.fit(X_train, y_train)

# L1-регуляризация
model_l1 = LogisticRegression(penalty='l1', solver='liblinear',
 ↪  C=1.0)
model_l1.fit(X_train, y_train)

# Без регуляризации
model_none = LogisticRegression(penalty=None, solver='lbfgs',
 ↪  max_iter=200)
```

```
model_none.fit(X_train, y_train)

print(f"Accuracy L2: {model_l2.score(X_test, y_test):.4f}")
print(f"Accuracy L1: {model_l1.score(X_test, y_test):.4f}")
print(f"Accuracy None: {model_none.score(X_test, y_test):.4f}")

# Параметр C контролирует регуляризацию
# C = 1/lambda, поэтому меньше C = больше регуляризация
for c in [0.1, 1.0, 10.0]:
    model = LogisticRegression(C=c, max_iter=200)
    model.fit(X_train, y_train)
    print(f"C={c}: Accuracy={model.score(X_test, y_test):.4f}")
```

14. К-ближайших соседей

14.1. Введение

K-Nearest Neighbors (KNN) — это непараметрический метод машинного обучения, используемый для задач классификации и регрессии. Алгоритм основан на **гипотезе компактности**: похожие объекты находятся близко друг к другу в пространстве признаков.

Основная идея

"Скажи мне, кто твой сосед, и я скажу, кто ты"

Алгоритм KNN классифицирует новый объект на основе классов его k ближайших соседей из обучающей выборки:

- Классификация: класс определяется голосованием (мода среди соседей)
- Регрессия: значение определяется усреднением соседей

Преимущества

- Простота: легко понять и реализовать
- Непараметричность: не делает предположений о распределении данных
- Адаптивность: автоматически адаптируется к локальным паттернам
- Универсальность: работает для классификации и регрессии
- Обучение: фаза обучения отсутствует ("ленивое обучение")

Недостатки

- Вычислительная сложность: медленное предсказание ($O(n \cdot d)$)
- Память: нужно хранить всю обучающую выборку
- Проклятие размерности: плохо работает в высоких размерностях
- Чувствительность к масштабу: требуется нормализация признаков
- Чувствительность к шуму: выбросы сильно влияют на результат

14.2. Алгоритм классификации

Обучение:

1. Запомнить всю обучающую выборку $\{(x_i, y_i)\}_{i=1}^{m}$
2. Задать параметр k (количество соседей)

Предсказание для нового объекта x_{new}:

1. Вычислить расстояние от x_{new} до всех объектов обучающей выборки
2. Выбрать k ближайших соседей
3. Классификация: найти класс, который встречается чаще всего среди k соседей (мода)

4. Регрессия: вычислить среднее значение целевой переменной среди k соседей

Задача: классифицировать новый объект (зелёный кружок).
Данные:

- Красные треугольники — класс A
- Синие квадраты — класс B

Если $k = 3$:

- 3 ближайших соседа: 2 треугольника, 1 квадрат
- Класс: A (большинство)

Если $k = 5$:

- 5 ближайших соседей: 2 треугольника, 3 квадрата
- Класс: B (большинство)

Вывод: выбор k критически важен.

14.3. Метрики расстояния

14.3.1 Евклидово расстояние

Формула:

$$d(x, y) = \sqrt{\sum_{j=1}^{n} (x_j - y_j)^2} = \|x - y\|_2$$

Применение: наиболее популярная метрика, подходит для непрерывных признаков.
Пример:

$$d([1, 2], [4, 6]) = \sqrt{(1-4)^2 + (2-6)^2} = \sqrt{9 + 16} = 5$$

14.3.2 Манхэттенское расстояние (L1)

Формула:

$$d(x, y) = \sum_{j=1}^{n} |x_j - y_j| = \|x - y\|_1$$

Применение: когда движение возможно только вдоль осей (как по улицам Манхэттена).
Пример:

$$d([1, 2], [4, 6]) = |1-4| + |2-6| = 3 + 4 = 7$$

14.3.3 Расстояние Минковского

Формула:

$$d(x, y) = \left(\sum_{j=1}^{n} |x_j - y_j|^p \right)^{1/p}$$

Частные случаи:

- $p = 1$: Манхэттенское расстояние
- $p = 2$: Евклидово расстояние
- $p \to \infty$: расстояние Чебышёва $\max_j |x_j - y_j|$

14.3.4 Расстояние Хэмминга

Формула:

$$d(x, y) = \sum_{j=1}^{n} \mathcal{J}(x_j \neq y_j)$$

где $\mathcal{J}$ — индикаторная функция.
Применение: для категориальных или бинарных признаков.
Пример:

$$d([0, 1, 1, 0], [0, 0, 1, 1]) = 0 + 1 + 0 + 1 = 2$$

14.3.5 Косинусное сходство

Формула:

$$\text{similarity}(x, y) = \frac{x \cdot y}{\|x\|\|y\|} = \frac{\sum x_j y_j}{\sqrt{\sum x_j^2}\sqrt{\sum y_j^2}}$$

Расстояние:

$$d(x, y) = 1 - \text{similarity}(x, y)$$

Применение: для текстов (TF-IDF векторов), когда важно направление, а не длина.

14.4. Выбор параметра k

14.4.1 Влияние k

Малое k (например, $k = 1$):

+ Гибкая граница решения
+ Хорошо улавливает локальные паттерны
– Чувствительность к шуму и выбросам
– Переобучение (overfitting)

Большое k (например, $k = 100$):

+ Сглаживание границ
+ Устойчивость к шуму
– Потеря локальных паттернов
– Недообучение (underfitting)

14.4.2 Правило выбора

Эмпирическое правило:

$$k \approx \sqrt{m}$$

где m — размер обучающей выборки.
Для классификации: выбирайте **нечётное** k, чтобы избежать ничьей при голосовании.

14.4.3 Подбор через кросс-валидацию

Алгоритм:

1. Задайте диапазон значений k: например, $k \in \{1, 3, 5, 7, \dots, 21\}$
2. Для каждого k вычислите средний score по кросс-валидации
3. Выберите k с максимальным score

```python
from sklearn.model_selection import cross_val_score
import numpy as np

k_values = range(1, 31, 2)  # нечётные от 1 до 29
scores = []

for k in k_values:
    knn = KNeighborsClassifier(n_neighbors=k)
    score = cross_val_score(knn, X_train, y_train, cv=5,
                            scoring='accuracy').mean()
    scores.append(score)

best_k = k_values[np.argmax(scores)]
print(f"Лучшее k: {best_k}")
```

14.5. Взвешенное голосование

14.5.1 Проблема стандартного KNN

При обычном голосовании все k соседей имеют равный вес. Но **ближние соседи** должны иметь больше влияния, чем дальние.

14.5.2 Взвешивание по расстоянию

Идея: вес соседа обратно пропорционален расстоянию.
Формула для классификации:

$$\hat{y} = \arg\max_c \sum_{i \in \text{kNN}} w_i \cdot \mathcal{I}(y_i = c)$$

где вес:

$$w_i = \frac{1}{d(x, x_i)^2}$$

или просто:

$$w_i = \frac{1}{d(x, x_i)}$$

Для регрессии:

$$\hat{y} = \frac{\sum_{i \in \text{kNN}} w_i \cdot y_i}{\sum_{i \in \text{kNN}} w_i}$$

Реализация в sklearn:

```python
from sklearn.neighbors import KNeighborsClassifier

# weights='uniform' — равные веса (по умолчанию)
knn_uniform = KNeighborsClassifier(n_neighbors=5,
 ↪  weights='uniform')

# weights='distance' — взвешивание по расстоянию
knn_weighted = KNeighborsClassifier(n_neighbors=5,
 ↪  weights='distance')

knn_weighted.fit(X_train, y_train)
```

14.6. Нормализация признаков

14.6.1 Почему важна нормализация

KNN использует **расстояние**, поэтому масштаб признаков критически важен.
Пример:

- Признак 1 (возраст): от 20 до 80 (диапазон: 60)
- Признак 2 (доход): от 20,000 до 200,000 (диапазон: 180,000)

Без нормализации признак 2 будет доминировать в вычислении расстояния.

14.7. Реализация

14.7.1 Полный пример

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score, classification_report
from sklearn.metrics import confusion_matrix

# Загрузка данных
iris = load_iris()
X, y = iris.data, iris.target

# Разделение
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=42, stratify=y
)

# Нормализация
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

```

# Обучение KNN
knn = KNeighborsClassifier(
    n_neighbors=5,
    weights='distance',  # взвешивание по расстоянию
    metric='euclidean'   # евклидова метрика
)
knn.fit(X_train_scaled, y_train)

# Предсказание
y_pred = knn.predict(X_test_scaled)

# Оценка
print(f"Accuracy: {accuracy_score(y_test, y_pred):.4f}")
print("\nClassification Report:")
print(classification_report(y_test, y_pred,
                            target_names=iris.target_names))

# Матрица ошибок
cm = confusion_matrix(y_test, y_pred)
print("\nConfusion Matrix:")
print(cm)

# Вероятности классов
y_proba = knn.predict_proba(X_test_scaled)
print("\nПримеры вероятностей (первые 3):")
for i in range(3):
    print(f"Объект {i}: {y_proba[i]}, класс:
     ↪  {iris.target_names[y_pred[i]]}")
```

14.7.2 Визуализация границ решения

```
from matplotlib.colors import ListedColormap

def plot_decision_boundary(X, y, model, title):
    # Используем только 2 признака для визуализации
    X_vis = X[:, :2]

    h = 0.02  # шаг сетки
    x_min, x_max = X_vis[:, 0].min() - 1, X_vis[:, 0].max() + 1
    y_min, y_max = X_vis[:, 1].min() - 1, X_vis[:, 1].max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                         np.arange(y_min, y_max, h))

    # Обучаем на 2 признаках
    model_2d = KNeighborsClassifier(n_neighbors=5)
    model_2d.fit(X_vis, y)

    # Предсказываем для всей сетки
    Z = model_2d.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)

    # Отрисовка
```

```
    cmap_light = ListedColormap(['#FFAAAA', '#AAFFAA', '#AAAAFF'])
    cmap_bold = ListedColormap(['#FF0000', '#00FF00', '#0000FF'])

    plt.figure(figsize=(8, 6))
    plt.contourf(xx, yy, Z, alpha=0.4, cmap=cmap_light)
    plt.scatter(X_vis[:, 0], X_vis[:, 1], c=y, cmap=cmap_bold,
                edgecolor='k', s=50)
    plt.xlabel('Feature 1')
    plt.ylabel('Feature 2')
    plt.title(title)
    plt.show()

# Визуализация
plot_decision_boundary(X_train_scaled, y_train, knn,
                       'KNN Decision Boundary (k=5)')
```

14.8. KNN для регрессии

Для регрессии предсказанное значение — это **среднее** (или взвешенное среднее) целевых значений k ближайших соседей:

$$\hat{y} = \frac{1}{k} \sum_{i \in \text{kNN}} y_i$$

или с весами:

$$\hat{y} = \frac{\sum_{i \in \text{kNN}} w_i \cdot y_i}{\sum_{i \in \text{kNN}} w_i}$$

Пример

```
from sklearn.neighbors import KNeighborsRegressor
from sklearn.datasets import make_regression
from sklearn.metrics import mean_squared_error, r2_score

# Генерируем данные
X, y = make_regression(n_samples=100, n_features=5, noise=10,
                       random_state=42)

# Разделяем
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=42
)

# Нормализуем
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Обучаем KNN регрессор
knn_reg = KNeighborsRegressor(
    n_neighbors=5,
    weights='distance'
)
```

```
knn_reg.fit(X_train_scaled, y_train)

# Предсказываем
y_pred = knn_reg.predict(X_test_scaled)

# Оцениваем
print(f"R² score: {r2_score(y_test, y_pred):.4f}")
print(f"MSE: {mean_squared_error(y_test, y_pred):.2f}")
print(f"RMSE: {np.sqrt(mean_squared_error(y_test, y_pred)):.2f}")
```

14.9. Оптимизация KNN

14.9.1 Проблема вычислительной сложности

Наивная реализация:

- Для каждого нового объекта нужно вычислить расстояние до всех m обучающих объектов
- Сложность: $O(m \cdot d)$, где d — размерность
- Для больших датасетов это очень медленно.

14.9.2 KD-Tree

Идея: построить дерево для быстрого поиска ближайших соседей.
Сложность:

- Построение: $O(m \log m)$
- Поиск: $O(\log m)$ в среднем

Ограничение: плохо работает для $d > 20$ (проклятие размерности).

```
from sklearn.neighbors import KNeighborsClassifier

# algorithm='kd_tree' — использование KD-Tree
knn = KNeighborsClassifier(n_neighbors=5, algorithm='kd_tree')
knn.fit(X_train, y_train)
```

14.9.3 Ball Tree

Идея: похоже на KD-Tree, но использует гиперсферы вместо гиперплоскостей.
Преимущество: лучше работает в высоких размерностях.

```
knn = KNeighborsClassifier(n_neighbors=5, algorithm='ball_tree')
knn.fit(X_train, y_train)
```

14.9.4 Approximate Nearest Neighbors

Для очень больших данных используют приближённые методы:

- Annoy (Spotify)
- FAISS (Facebook)
- HNSW (Hierarchical Navigable Small World)

Жертвуют точностью ради скорости (поиск за $O(\log m)$).

14.10. Radius Neighbors

14.10.1 Альтернатива KNN

Вместо фиксированного k используют **радиус** r: учитываются все соседи в радиусе r.
Формула:

$$\text{Neighbors}(x) = \{x_i : d(x, x_i) \leq r\}$$

Преимущество: контролируем похожесть соседей.
Недостаток: количество соседей может сильно варьироваться.

```python
from sklearn.neighbors import RadiusNeighborsClassifier

rnn = RadiusNeighborsClassifier(radius=1.0)
rnn.fit(X_train_scaled, y_train)
y_pred = rnn.predict(X_test_scaled)
```

14.11. Проклятие размерности

Проблема
В высоких размерностях ($d \gg 10$) расстояния между всеми точками становятся примерно одинаковыми.
Пример: в 100-мерном пространстве почти все точки находятся на примерно одинаковом расстоянии друг от друга.
Следствия

- Понятие “близости” теряет смысл
- KNN деградирует до случайного угадывания
- Нужно экспоненциально больше данных для покрытия пространства

Решения:

1. Отбор признаков: использовать только важные признаки
2. Понижение размерности
3. Feature engineering: создать более информативные признаки
4. Другие модели: использовать алгоритмы, не зависящие от расстояний

14.12. Практические рекомендации

Когда использовать KNN?
Хорошо работает:

- Малая размерность: $d < 20$
- Нелинейные границы: KNN может моделировать сложные границы
- Малые данные: не требует большой выборки для обучения
- Быстрое прототипирование: легко реализовать baseline

- Системы рекомендаций: поиск похожих пользователей/товаров

Плохо работает:

- Большие данные: медленное предсказание
- Высокая размерность: проклятие размерности
- Несбалансированные классы: доминирующий класс побеждает
- Категориальные признаки: нужно кодирование

14.13. Заключение

- KNN — простой и интуитивный метод классификации и регрессии
- Основан на расстоянии в пространстве признаков
- Ленивое обучение: модель запоминает данные, а не учится
- Критически важна нормализация признаков
- Выбор k влияет на баланс между переобучением и недообучением
- Хорошо работает для малой размерности, плохо для высокой
- Медленное предсказание - главный недостаток

15. Дерево решений

15.1. Введение

Дерево решений (Decision Tree) — это непараметрический метод обучения с учителем, используемый для задач классификации и регрессии. Модель представляет собой иерархическую древовидную структуру, состоящую из решающих правил вида "Если..., то...".

Дерево решений рекурсивно разбивает пространство признаков на области, в каждой из которых делается простое предсказание (класс или значение).

Аналогия: дерево решений работает как блок-схема (flowchart), где в каждом узле задаётся вопрос о признаке, и в зависимости от ответа мы переходим к следующему узлу.

Структура дерева:

- Корневой узел (root node) — начальный узел дерева, содержащий все данные
- Внутренние узлы (internal nodes) — узлы с решающими правилами
- Листья (leaf nodes) — конечные узлы с предсказаниями
- Ветви (branches) — рёбра, соединяющие узлы
- Решающее правило — условие в узле (например, "Возраст < 30?")

Преимущества:

- Интерпретируемость: легко понять и объяснить
- Визуализация: можно нарисовать дерево
- Нелинейности: моделирует сложные зависимости
- Смешанные данные: работает с числовыми и категориальными признаками
- Не требует нормализации: масштаб признаков не важен
- Feature importance: показывает важность признаков

Недостатки:

- Переобучение: склонность к overfitting
- Нестабильность: малое изменение данных может сильно изменить дерево
- Жадность: локально-оптимальные разбиения могут быть глобально неоптимальными
- Несбалансированные данные: смещение в сторону доминирующих классов

15.2. Принцип работы

15.2.1 Алгоритм построения

Рекурсивный алгоритм (top-down, greedy):

1. Начать с корневого узла, содержащего всю обучающую выборку
2. Для текущего узла:

- Если узел “чистый” (все объекты одного класса) сделать листом
- Если достигнута максимальная глубина сделать листом
- Иначе:
  (a) Найти **лучшее разбиение** по одному из признаков
  (b) Разделить данные на подмножества по этому признаку
  (c) Создать дочерние узлы

3. Рекурсивно применить алгоритм к каждому дочернему узлу

15.2.2 Критерии разбиения

Ключевой вопрос: как выбрать лучший признак и порог для разбиения?
Цель: создать наиболее “чистые” подмножества.
Существует несколько критериев:

- Энтропия и прирост информации (Information Gain)
- Индекс Джини (Gini Index)
- Среднеквадратичная ошибка (MSE) для регрессии

15.3. Энтропия и прирост информации

15.3.1 Энтропия

Энтропия — мера неопределённости или “беспорядка” в данных.
Формула для набора данных S с K классами:

$$\boxed{H(S) = -\sum_{k=1}^{K} p_k \log_2 p_k}$$

где p_k — доля объектов класса k в S.
Свойства:

- $H(S) = 0$ — все объекты одного класса (полная определённость)
- $H(S) = \log_2 K$ — равномерное распределение классов (максимальная неопределённость)
- $0 \leq H(S) \leq \log_2 K$

Пример: бинарная классификация с $p_1 = 0.5, p_2 = 0.5$:

$$H(S) = -0.5 \log_2 0.5 - 0.5 \log_2 0.5 = -0.5 \cdot (-1) - 0.5 \cdot (-1) = 1 \text{ бит}$$

15.3.2 Условная энтропия

После разбиения по признаку A получаем подмножества $S_1, S_2, \ldots, S_v$. **Условная энтропия**:

$$H(S|A) = \sum_{i=1}^{v} \frac{|S_i|}{|S|} H(S_i)$$

где $|S_i|$ — размер подмножества S_i.

15.3.3 Прирост информации

Information Gain (IG) — насколько уменьшается энтропия после разбиения:

$$\boxed{IG(S, A) = H(S) - H(S|A)}$$

Цель: выбрать признак A с **максимальным приростом** информации.

15.3.4 Пример расчёта

Данные: 14 студентов, 9 сдали экзамен (+), 5 не сдали ( ).
Энтропия до разбиения:

$$H(S) = -\frac{9}{14}\log_2\frac{9}{14} - \frac{5}{14}\log_2\frac{5}{14} \approx 0.940 \text{ бита}$$

Разбиение по признаку "Учился" (Да/Нет):

- S_1 (Да): 7 студентов, 6+, 1 $\quad H(S_1) = -\frac{6}{7}\log_2\frac{6}{7} - \frac{1}{7}\log_2\frac{1}{7} \approx 0.592$
- S_2 (Нет): 7 студентов, 3+, 4 $\quad H(S_2) = -\frac{3}{7}\log_2\frac{3}{7} - \frac{4}{7}\log_2\frac{4}{7} \approx 0.985$

Условная энтропия:

$$H(S|\text{Учился}) = \frac{7}{14} \cdot 0.592 + \frac{7}{14} \cdot 0.985 \approx 0.789$$

Прирост информации:

$$IG(S, \text{Учился}) = 0.940 - 0.789 = 0.151 \text{ бита}$$

Вывод: разбиение уменьшило энтропию на 0.151 бита.

15.3.5 Gain Ratio

Проблема IG: предпочитает признаки с большим количеством значений.
Решение: нормализовать на собственную энтропию признака:

$$\text{Gain Ratio}(S, A) = \frac{IG(S, A)}{\text{SplitInfo}(S, A)}$$

где:

$$\text{SplitInfo}(S, A) = -\sum_{i=1}^{v} \frac{|S_i|}{|S|}\log_2\frac{|S_i|}{|S|}$$

Использование: алгоритм C4.5.

15.4. Индекс Джини

15.4.1 Определение

Индекс Джини (Gini Impurity) — альтернатива энтропии, мера "примеси" в узле.
Формула:

$$\boxed{Gini(S) = 1 - \sum_{k=1}^{K} p_k^2}$$

Интерпретация: вероятность неправильно классифицировать случайно выбранный объект, если класс присваивается случайно согласно распределению в S.

Свойства:

- $Gini(S) = 0$ — все объекты одного класса (чистый узел)
- $Gini(S) = 1 - \frac{1}{K}$ — равномерное распределение (максимальная примесь)
- Для бинарной классификации: $Gini(S) \in [0, 0.5]$

15.4.2 Gini Index после разбиения

$$Gini(S, A) = \sum_{i=1}^{v} \frac{|S_i|}{|S|} Gini(S_i)$$

Цель: выбрать признак A с **минимальным Gini Index**.

15.4.3 Пример

Данные: 9+, 5 (всего 14).
Gini до разбиения:

$$Gini(S) = 1 - \left(\frac{9}{14}\right)^2 - \left(\frac{5}{14}\right)^2 \approx 1 - 0.413 - 0.128 = 0.459$$

Разбиение по "Учился":

- S_1 (Да): 6+, 1 $\quad Gini(S_1) = 1 - \left(\frac{6}{7}\right)^2 - \left(\frac{1}{7}\right)^2 \approx 0.245$
- S_2 (Нет): 3+, 4 $\quad Gini(S_2) = 1 - \left(\frac{3}{7}\right)^2 - \left(\frac{4}{7}\right)^2 \approx 0.490$

Gini Index:

$$Gini(S, \text{Учился}) = \frac{7}{14} \cdot 0.245 + \frac{7}{14} \cdot 0.490 = 0.368$$

Уменьшение примеси:

$$\Delta Gini = 0.459 - 0.368 = 0.091$$

15.5. Разбиение по числовым признакам

15.5.1 Проблема

Для категориальных признаков разбиение естественное (по значениям). Но как разбивать числовые признаки?

15.5.2 Алгоритм

1. Отсортировать значения признака: $x_1 \leq x_2 \leq \ldots \leq x_m$
2. Рассмотреть все возможные пороги между соседними значениями:

$$t_i = \frac{x_i + x_{i+1}}{2}, \quad i = 1, \ldots, m-1$$

3. Для каждого порога t вычислить критерий разбиения:
   - $S_{\text{left}} : x \leq t$
   - $S_{\text{right}} : x > t$
4. Выбрать порог t^* с максимальным IG (или минимальным Gini)

Сложность: $O(m \log m)$ для сортировки + $O(m)$ для перебора порогов.

15.5.3 Пример

Признак: Возраст = [22, 25, 30, 35, 40]
Классы: [+, +, , ,]
Пороги: $t \in \{23.5, 27.5, 32.5, 37.5\}$
Для каждого порога вычисляем IG и выбираем лучший.

15.6. Алгоритмы построения деревьев

15.6.1 ID3 (Iterative Dichotomiser 3)

Автор: Ross Quinlan (1986)
Особенности:

- Использует **прирост информации** (Entropy + IG)
- Только **категориальные** признаки
- Не поддерживает обрезку (pruning)
- Не обрабатывает пропущенные значения

15.6.2 C4.5

Автор: Ross Quinlan (1993), улучшенная версия ID3
Особенности:

- Использует **Gain Ratio** (нормализованный IG)
- Поддержка **числовых** признаков
- Обрезка дерева (pruning)
- Обработка **пропущенных значений**
- Генерация **правил** из дерева

15.6.3 CART (Classification and Regression Trees)

Авторы: Breiman et al. (1984)
Особенности:

- Использует **Индекс Джини** для классификации
- Использует **MSE** для регрессии
- Строит **бинарные** деревья (всегда 2 ветви)
- Поддержка числовых и категориальных признаков
- Обрезка с минимальной стоимостью-сложностью (cost-complexity pruning)

Используется в scikit-learn: `DecisionTreeClassifier` и `DecisionTreeRegressor`.

15.7. Дерево регрессии

15.7.1 Отличия от классификации

Классификация: предсказываем класс (дискретная переменная).
Регрессия: предсказываем непрерывное значение.

15.7.2 Критерий разбиения: MSE

Среднеквадратичная ошибка (Mean Squared Error):

$$MSE(S) = \frac{1}{|S|} \sum_{i \in S} (y_i - \bar{y})^2$$

где $\bar{y} = \frac{1}{|S|} \sum_{i \in S} y_i$ — среднее значение в узле.
После разбиения:

$$MSE(S, A) = \frac{|S_{\text{left}}|}{|S|} MSE(S_{\text{left}}) + \frac{|S_{\text{right}}|}{|S|} MSE(S_{\text{right}})$$

Цель: минимизировать MSE.

15.7.3 Предсказание в листе

В листе предсказываем **среднее значение**:

$$\hat{y} = \frac{1}{|S_{\text{leaf}}|} \sum_{i \in S_{\text{leaf}}} y_i$$

15.7.4 Альтернативные критерии

- MAE (Mean Absolute Error): $\frac{1}{|S|} \sum |y_i - \text{median}(y)|$
- Friedman MSE: улучшенная версия MSE

15.8. Переобучение и обрезка

15.8.1 Проблема переобучения

Без ограничений дерево может расти до тех пор, пока каждый лист содержит один объект. Такое дерево **идеально** предсказывает обучающую выборку, но **плохо обобщает** на новые данные.

Признаки переобучения:

- Очень глубокое дерево
- Много листьев с малым количеством объектов
- Высокая точность на train, низкая на test

15.8.2 Предварительная обрезка (Pre-pruning)

Идея: остановить рост дерева раньше, не дожидаясь полной чистоты листьев.
Критерии остановки:

- max_depth: максимальная глубина дерева
- min_samples_split: минимальное число объектов для разбиения узла
- min_samples_leaf: минимальное число объектов в листе
- max_leaf_nodes: максимальное число листьев
- min_impurity_decrease: минимальное уменьшение примеси

Реализация в sklearn:

```
from sklearn.tree import DecisionTreeClassifier

tree = DecisionTreeClassifier(
    max_depth=5,                # не глубже 5 уровней
    min_samples_split=20,       # минимум 20 объектов для разбиения
    min_samples_leaf=10,        # минимум 10 объектов в листе
    max_leaf_nodes=30           # максимум 30 листьев
)
tree.fit(X_train, y_train)
```

15.8.3 Последующая обрезка (Post-pruning)

Идея: сначала построить полное дерево, затем обрезать лишние ветви.

15.8.4 Обрезка с минимальной стоимостью-сложностью

Cost-Complexity Pruning (CCP):
Для поддерева T определяем функцию стоимости:

$$R_\alpha(T) = R(T) + \alpha|T|$$

где:

- $R(T)$ — ошибка на обучающей выборке
- $|T|$ — количество листьев в дереве
- $\alpha \geq 0$ — параметр сложности

Алгоритм:

1. Построить полное дерево
2. Для каждого α найти оптимальное поддерево T_α
3. Выбрать α через кросс-валидацию

Реализация в sklearn:

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

# Строим полное дерево
tree = DecisionTreeClassifier(random_state=42)
tree.fit(X_train, y_train)

# Получаем путь обрезки
path = tree.cost_complexity_pruning_path(X_train, y_train)
ccp_alphas = path.ccp_alphas
impurities = path.impurities

# Находим лучший alpha через кросс-валидацию
scores = []
for alpha in ccp_alphas:
    tree_pruned = DecisionTreeClassifier(ccp_alpha=alpha,
    ↪  random_state=42)
    score = cross_val_score(tree_pruned, X_train, y_train,
    ↪  cv=5).mean()
    scores.append(score)

best_alpha = ccp_alphas[np.argmax(scores)]
print(f"Лучший alpha: {best_alpha:.4f}")

# Обучаем с лучшим alpha
final_tree = DecisionTreeClassifier(ccp_alpha=best_alpha,
 ↪  random_state=42)
final_tree.fit(X_train, y_train)
```

15.9. Реализация в scikit-learn

15.9.1 Классификация

```python
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score, classification_report
from sklearn import tree
import matplotlib.pyplot as plt

# Загружаем данные
iris = load_iris()
X, y = iris.data, iris.target

# Разделяем
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=42
)

# Создаём и обучаем дерево
clf = DecisionTreeClassifier(
```

```
    criterion='gini',           # или 'entropy'
    max_depth=3,                # ограничение глубины
    min_samples_split=5,
    min_samples_leaf=2,
    random_state=42
)
clf.fit(X_train, y_train)

# Предсказываем
y_pred = clf.predict(X_test)

# Оцениваем
print(f"Accuracy: {accuracy_score(y_test, y_pred):.4f}")
print("\nClassification Report:")
print(classification_report(y_test, y_pred,
                            target_names=iris.target_names))

# Важность признаков
print("\nFeature Importances:")
for name, importance in zip(iris.feature_names,
 ↪  clf.feature_importances_):
    print(f"{name}: {importance:.4f}")
```

15.9.2 Визуализация дерева

```
import matplotlib.pyplot as plt
from sklearn import tree

# Визуализация дерева
plt.figure(figsize=(20, 10))
tree.plot_tree(clf,
               feature_names=iris.feature_names,
               class_names=iris.target_names,
               filled=True,
               rounded=True,
               fontsize=10)
plt.title("Decision Tree Visualization")
plt.show()

# Экспорт в текстовый формат
from sklearn.tree import export_text
tree_rules = export_text(clf, feature_names=iris.feature_names)
print(tree_rules)
```

15.9.3 Регрессия

```
from sklearn.tree import DecisionTreeRegressor
from sklearn.datasets import make_regression
from sklearn.metrics import mean_squared_error, r2_score
import numpy as np

# Генерируем данные
```

```python
X, y = make_regression(n_samples=100, n_features=4, noise=10,
                       random_state=42)

# Разделяем
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=42
)

# Обучаем дерево регрессии
reg = DecisionTreeRegressor(
    criterion='squared_error',  # или 'absolute_error'
    max_depth=5,
    min_samples_split=10,
    min_samples_leaf=5,
    random_state=42
)
reg.fit(X_train, y_train)

# Предсказываем
y_pred = reg.predict(X_test)

# Оцениваем
print(f"R² score: {r2_score(y_test, y_pred):.4f}")
print(f"MSE: {mean_squared_error(y_test, y_pred):.2f}")
print(f"RMSE: {np.sqrt(mean_squared_error(y_test, y_pred)):.2f}")

# Важность признаков
print("\nFeature Importances:")
for i, importance in enumerate(reg.feature_importances_):
    print(f"Feature {i}: {importance:.4f}")
```

15.10. Feature Importance

Feature Importance — мера того, насколько каждый признак полезен для построения дерева.

15.10.1 Вычисление

Для признака f:

$$\text{Importance}(f) = \sum_{t \in \text{splits on } f} \frac{|S_t|}{|S|} \Delta\text{Impurity}_t$$

где:

- Сумма по всем узлам t, где происходит разбиение по признаку f
- $|S_t|$ — количество объектов в узле t
- $\Delta\text{Impurity}_t$ — уменьшение примеси в узле t

Нормализация: $\sum_f \text{Importance}(f) = 1$

15.10.2 Использование

```
# Получаем важность признаков
importances = clf.feature_importances_

# Сортируем
indices = np.argsort(importances)[::-1]

# Визуализация
plt.figure(figsize=(10, 6))
plt.title("Feature Importances")
plt.bar(range(X.shape[1]), importances[indices])
plt.xticks(range(X.shape[1]),
           [iris.feature_names[i] for i in indices],
           rotation=45)
plt.show()
```

15.11. Когда использовать деревья решений

Хорошо работают:

- Интерпретируемость важна: нужно объяснить решения
- Смешанные данные: числовые + категориальные признаки
- Нелинейные зависимости: сложные взаимодействия признаков
- Не требуется предобработка: не нужна нормализация
- Автоматический отбор признаков: дерево само выбирает важные

Плохо работают:

- Линейные зависимости: дерево будет делать много разбиений
- Экстраполяция: не может предсказывать за пределами обучающих данных
- Нестабильность: малое изменение данных большое изменение дерева
- Переобучение: без ограничений легко переобучается

15.12. Заключение

- Деревья решений — **интуитивный и интерпретируемый** метод
- Строятся **рекурсивным разбиением** пространства признаков
- Критерии разбиения: **Энтропия (IG)** или **Индекс Джини**
- Основные алгоритмы: **ID3, C4.5, CART**
- Склонны к **переобучению** нужна обрезка
- Показывают **важность признаков**
- Хороши для **интерпретируемых** моделей
- Плохи для **больших данных** и **стабильности**

16. Ансамбли. Случайный лес

16.1. Введение: ансамбли моделей

Определение 16.1. Ансамбль моделей

Ансамбль — это комбинация нескольких моделей, где итоговое предсказание получается путём объединения предсказаний отдельных моделей.

Главная идея: "Мудрость толпы" — множество простых моделей вместе могут превзойти одну сложную модель.

Метафора: Представьте, что вам нужно оценить количество конфет в банке:

- Один человек может сильно ошибиться (переоценить или недооценить)
- Среднее из 100 оценок разных людей обычно близко к истине
- Ошибки отдельных людей компенсируют друг друга

Проблема одиночных моделей

Одна модель (линейная регрессия, одно дерево решений) может:

- Недообучиться (высокое смещение) — слишком простая модель
- Переобучиться (высокая дисперсия) — слишком сложная модель
- Быть нестабильной — малое изменение данных приводит к большому изменению модели

Пример нестабильности: одно дерево решений очень чувствительно к данным. Если добавить или убрать несколько примеров, структура дерева может сильно измениться.

Сравнение подходов:

- Bagging: параллельное обучение $\rightarrow$ быстро, легко распараллелить, не переобучается
- Boosting: последовательное обучение $\rightarrow$ медленнее, но точнее, риск переобучения
- Stacking: комбинирование разных моделей $\rightarrow$ максимальная гибкость, сложная настройка

Аналогии из жизни:

- Bagging: спросить мнение у 100 случайных людей о количестве конфет в банке и усреднить — среднее будет точнее, чем любой один ответ
- Boosting: решать контрольную работу несколько раз, каждый раз фокусируясь на тех задачах, где ошиблись в прошлый раз
- Stacking: попросить эксперта (врач), генералиста (семейный врач) и специалиста (кардиолог) поставить диагноз, а затем главный врач принимает финальное решение на основе их мнений

Bias-Variance Trade-off

Общая ошибка модели раскладывается на три компоненты:

$$\text{Error} = \text{Bias}^2 + \text{Variance} + \text{Irreducible Error}$$

- Bias (смещение) — систематическая ошибка модели, недообучение

- Variance (дисперсия) — чувствительность модели к данным, переобучение
- Irreducible Error — шум в данных, неустранимая ошибка

Как ансамбли помогают:

- Bagging уменьшает **дисперсию** (variance) → меньше переобучение
- Boosting уменьшает **смещение** (bias) → лучше подгонка к данным

16.2. Bagging (Bootstrap Aggregating)

Что такое Bootstrap
Bootstrap — это статистический метод ресамплирования данных **с возвращением**.
Алгоритм:

1. Из исходной выборки размера m делаем B новых выборок размера m с возвращением
2. Каждая новая выборка содержит примерно 63.2% уникальных объектов из исходной
3. $\approx$ 36.8% объектов не попадают в выборку (out-of-bag, OOB)

Математика: вероятность, что объект НЕ попадёт в bootstrap-выборку:

$$P(\text{not selected}) = \left(1 - \frac{1}{m}\right)^m \xrightarrow{m\to\infty} \frac{1}{e} \approx 0.368$$

Пример: есть 5 объектов $\{A, B, C, D, E\}$. Bootstrap-выборка с возвращением может быть $\{A, A, C, D, E\}$ — объект A попался дважды, объект B не попал совсем.
Bootstrap Aggregating (Bagging):

1. Создать B bootstrap-выборок из обучающих данных
2. Обучить базовую модель (обычно дерево решений) на каждой выборке
3. Объединить предсказания:
   - Классификация: голосование большинством
   - Регрессия: усреднение

Формула для классификации:

$$\hat{y} = \text{mode}\left(\hat{y}_1, \hat{y}_2, \ldots, \hat{y}_B\right)$$

Формула для регрессии:

$$\hat{y} = \frac{1}{B}\sum_{b=1}^{B} \hat{y}_b$$

Почему Bagging работает?
Снижение дисперсии:

- Одно дерево имеет высокую дисперсию (нестабильно)
- Усреднение B деревьев снижает дисперсию
- Точность улучшается.

Математически: если модели независимы и имеют дисперсию σ^2, то среднее имеет дисперсию $\frac{\sigma^2}{B}$.

Проблема: на практике модели не независимы (обучены на похожих данных), поэтому дисперсия уменьшается не в B раз, а меньше:

$$\text{Var}(\text{среднее}) = \rho\sigma^2 + \frac{1-\rho}{B}\sigma^2$$

где ρ — корреляция между моделями. Чем меньше ρ, тем лучше.

16.3. Random Forest

Случайный лес (Random Forest, RF) — это ансамблевый метод машинного обучения, основанный на построении множества деревьев решений и объединении их предсказаний.

Основная идея

Random Forest решает главную проблему деревьев решений — **переобучение** и **нестабильность**.

Random Forest = Bagging + Feature Randomness

Проблема стандартного Bagging

Bootstrap-выборки похожи друг на друга, поэтому деревья получаются **коррелированными**.

Пример: если есть один очень сильный признак, то все деревья будут разбиваться по нему в корне $\rightarrow$ деревья похожи $\rightarrow$ плохая диверсификация $\rightarrow$ корреляция ρ высокая.

Решение: случайный выбор признаков

Random Forest добавляет **случайность при выборе признаков**:

- При каждом разбиении узла случайно выбираем подмножество из m_{try} признаков
- Выбираем лучший признак только из этого подмножества
- Это де-коррелирует деревья. ($\rho \downarrow$)

Гиперпараметр m_{try} (`max_features` в sklearn):

- Классификация: $m_{try} \approx \sqrt{p}$, где p — количество признаков
- Регрессия: $m_{try} \approx \frac{p}{3}$

Крайние случаи:

- $m_{try} = p$: обычный bagging (нет случайности признаков, высокая корреляция)
- $m_{try} = 1$: максимальная случайность (может быть слишком слабые деревья)

16.4. Алгоритм Random Forest

Обучение:

1. Для $b = 1, \dots, B$:
   (a) Создать bootstrap-выборку D_b из D
   (b) Построить дерево T_b на D_b:
      - В каждом узле случайно выбрать m_{try} признаков
      - Выбрать лучший признак из этих m_{try} для split
      - Разбить узел

(c) Вырастить дерево до максимальной глубины (без обрезки)

Предсказание:

- Классификация: голосование большинством среди B деревьев
- Регрессия: среднее предсказаний B деревьев

16.5. Преимущества и недостатки Random Forest

Преимущества

- Высокая точность: один из лучших алгоритмов "из коробки"
- Устойчивость к переобучению: меньше overfitting, чем у одного дерева
- Работа с шумом: робастность к выбросам
- Не требует нормализации: масштаб признаков не важен
- Параллелизм: деревья обучаются независимо
- Feature importance: показывает важность признаков
- OOB error: встроенная валидация
- Работа с пропусками: может обрабатывать missing values

Недостатки

- Интерпретируемость: сложно объяснить, как работает лес
- Память: нужно хранить множество деревьев
- Скорость предсказания: медленнее, чем одно дерево
- Экстраполяция: не может предсказывать за пределами обучающих данных
- Размер модели: большой файл при сохранении

16.6. Out-of-Bag (OOB) Error

Что такое OOB

Для каждого bootstrap-дерева $\approx 36.8\%$ объектов не использовались при обучении. Эти объекты называются **out-of-bag** (OOB).

OOB Error как валидация

Идея: использовать OOB объекты для оценки качества модели **без отдельной тестовой выборки**.

Алгоритм:

1. Для каждого объекта i найти все деревья, которые НЕ использовали его при обучении
2. Предсказать $\hat{y}_i$ усреднением/голосованием только этих деревьев
3. Вычислить ошибку между $\hat{y}_i$ и y_i

OOB error:

$$\text{OOB error} = \frac{1}{m}\sum_{i=1}^{m} L(y_i, \hat{y}_i^{\text{OOB}})$$

Преимущества OOB

- Экономия данных: не нужно отделять тестовую выборку
- Эквивалентно CV: OOB $\approx$ leave-one-out cross-validation
- Бесплатно: вычисляется во время обучения
- Честная оценка: OOB объекты не участвовали в обучении

Реализация в sklearn

```python
from sklearn.ensemble import RandomForestClassifier

rf = RandomForestClassifier(
    n_estimators=100,
    oob_score=True,  # включить вычисление OOB error
    random_state=42
)

rf.fit(X, y)  # можно использовать ВСЕ данные
print(f"OOB Score: {rf.oob_score_:.4f}")
```

16.7. Feature Importance

Важность признаков в Random Forest

Random Forest автоматически вычисляет **важность каждого признака**.

Метод: для каждого признака суммируем уменьшение примеси (Gini/Entropy) во всех узлах всех деревьев, где этот признак используется.

Формула:

$$\text{Importance}(f) = \frac{1}{B}\sum_{b=1}^{B} \sum_{t \in T_b : \text{split on } f} \frac{|S_t|}{|D_b|} \Delta\text{Impurity}_t$$

Нормализация: $\sum_f \text{Importance}(f) = 1$

Реализация

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.ensemble import RandomForestClassifier
from sklearn.datasets import load_iris

# Данные
iris = load_iris()
X, y = iris.data, iris.target

# Обучаем
rf = RandomForestClassifier(n_estimators=100, random_state=42)
rf.fit(X, y)

```

```
# Важность признаков
importances = rf.feature_importances_
indices = np.argsort(importances)[::-1]

# Визуализация
plt.figure(figsize=(10, 6))
plt.title("Feature Importances")
plt.bar(range(X.shape[1]), importances[indices])
plt.xticks(range(X.shape[1]),
           [iris.feature_names[i] for i in indices],
           rotation=45)
plt.ylabel('Importance')
plt.show()

# Вывод
for i in indices:
    print(f"{iris.feature_names[i]}: {importances[i]:.4f}")
```

Permutation Importance

Альтернативный метод: измерить, насколько ухудшится качество, если случайно перемешать значения признака.

```
from sklearn.inspection import permutation_importance

# Вычисляем permutation importance
perm_importance = permutation_importance(
    rf, X, y, n_repeats=10, random_state=42
)

# Сортируем
perm_indices = np.argsort(perm_importance.importances_mean)[::-1]
print("Permutation Importance:")
for i in perm_indices:
    print(f"{iris.feature_names[i]}: "
          f"{perm_importance.importances_mean[i]:.4f} "
          f"± {perm_importance.importances_std[i]:.4f}")
```

16.8. Реализация в scikit-learn

Классификация

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, classification_report

# Загружаем данные
data = load_breast_cancer()
X, y = data.data, data.target

# Разделяем
X_train, X_test, y_train, y_test = train_test_split(
```

```
    X, y, test_size=0.3, random_state=42, stratify=y
)

# Создаём Random Forest
rf = RandomForestClassifier(
    n_estimators=100,          # количество деревьев
    max_depth=None,            # максимальная глубина (None = без
    ↪  ограничений)
    min_samples_split=2,       # минимум для разбиения узла
    min_samples_leaf=1,        # минимум в листе
    max_features='sqrt',       # sqrt(p) признаков в узле
    bootstrap=True,            # использовать bootstrap
    oob_score=True,            # вычислить OOB error
    n_jobs=-1,                 # параллелизм (все ядра)
    random_state=42
)

# Обучаем
rf.fit(X_train, y_train)

# Предсказываем
y_pred = rf.predict(X_test)
y_proba = rf.predict_proba(X_test)

# Оцениваем
print(f"Train Accuracy: {rf.score(X_train, y_train):.4f}")
print(f"Test Accuracy: {accuracy_score(y_test, y_pred):.4f}")
print(f"OOB Score: {rf.oob_score_:.4f}")
print("\nClassification Report:")
print(classification_report(y_test, y_pred,
                            target_names=data.target_names))
```

Регрессия

```
from sklearn.ensemble import RandomForestRegressor
from sklearn.datasets import load_diabetes
from sklearn.metrics import mean_squared_error, r2_score
import numpy as np

# Загружаем данные
diabetes = load_diabetes()
X, y = diabetes.data, diabetes.target

# Разделяем
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=42
)

# Random Forest Regressor
rf_reg = RandomForestRegressor(
    n_estimators=100,
    max_depth=None,
    max_features='sqrt',      # или 1/3 для регрессии
```

```
    bootstrap=True,
    oob_score=True,
    n_jobs=-1,
    random_state=42
)

# Обучаем
rf_reg.fit(X_train, y_train)

# Предсказываем
y_pred = rf_reg.predict(X_test)

# Оцениваем
print(f"Train R²: {rf_reg.score(X_train, y_train):.4f}")
print(f"Test R²: {r2_score(y_test, y_pred):.4f}")
print(f"OOB Score: {rf_reg.oob_score_:.4f}")
print(f"RMSE: {np.sqrt(mean_squared_error(y_test, y_pred)):.2f}")
```

16.9. Гиперпараметры и настройка

Основные гиперпараметры

Параметр	По умолчанию	Рекомендации
n_estimators	100	Чем больше, тем лучше (100-500). Не переобучится
max_features	'sqrt' (класс), 1/3 (регр)	sqrt(p) или log2(p)
max_depth	None	Ограничить для малых данных ($<$ 1000 примеров)
min_samples_split	2	Увеличить (5-10) для борьбы с overfitting
min_samples_leaf	1	Увеличить (2-5) для сглаживания
bootstrap	True	Всегда True для RF
oob_score	False	Включить для бесплатной валидации

Таблица 4: Гиперпараметры Random Forest

Подбор гиперпараметров

```
from sklearn.model_selection import RandomizedSearchCV

# Сетка параметров
param_dist = {
    'n_estimators': [100, 200, 300, 500],
    'max_features': ['sqrt', 'log2', None],
    'max_depth': [10, 20, 30, None],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4],
    'bootstrap': [True]
}

# Randomized Search
rf = RandomForestClassifier(random_state=42)
```

```
random_search = RandomizedSearchCV(
    rf,
    param_distributions=param_dist,
    n_iter=50,                # количество комбинаций
    cv=5,
    scoring='accuracy',
    n_jobs=-1,
    random_state=42,
    verbose=1
)

random_search.fit(X_train, y_train)
print(f"Лучшие параметры: {random_search.best_params_}")
print(f"Лучший score: {random_search.best_score_:.4f}")
```

16.10. Практические рекомендации

Хорошо работает:

- Общего назначения: отличный baseline для большинства задач
- Табличные данные: структурированные данные с признаками
- Нелинейные зависимости: сложные взаимодействия признаков
- Малая предобработка: не требует нормализации, масштабирования
- Feature importance: нужно понять важность признаков
- Устойчивость: данные с шумом и выбросами

Плохо работает:

- Очень высокая размерность: text, images (используйте deep learning)
- Экстраполяция: предсказания за пределами обучающих данных
- Интерпретируемость: если нужна полная прозрачность модели
- Онлайн-обучение: RF не поддерживает инкрементальное обучение
- Большие данные: для $> 1M$ примеров лучше LightGBM

16.11. Итоги

- **Ансамбли** объединяют множество моделей для улучшения качества
- **Bagging** уменьшает дисперсию через усреднение
- **Bootstrap** создаёт разнообразные обучающие выборки
- **Random Forest** = Bagging + случайность признаков
- Решает проблемы **переобучения и нестабильности** деревьев
- **OOB error** — бесплатная валидация

- **Feature importance** — автоматический отбор признаков
- Один из лучших **“из коробки”** алгоритмов
- Отличный **baseline** для большинства задач
- Требует **минимальной настройки**

17. Градиентные бустинги

17.1. Введение в Boosting

Определение 17.1. Boosting (бустинг)

Boosting — это ансамблевый метод, где модели обучаются **последовательно**, и каждая следующая модель исправляет ошибки предыдущих.

Главная идея: "Учимся на своих ошибках" — каждая новая модель фокусируется на примерах, где предыдущие модели ошиблись.

Отличие от Bagging:

- Bagging (Random Forest): модели обучаются **параллельно**, независимо друг от друга
- Boosting: модели обучаются **последовательно**, каждая зависит от предыдущей

Интуиция бустинга
Метафора: решение сложной задачи по математике

1. Шаг 1: вы решаете задачу, получаете 60% правильных ответов
2. Шаг 2: анализируете ошибки, учите только те темы, где ошиблись
3. Шаг 3: решаете задачу снова, теперь 80% правильных ответов
4. Шаг 4: снова анализируете ошибки, снова учите слабые места
5. Результат: после нескольких итераций вы решаете 95% задач.

Бустинг работает так же: каждая новая модель "учится" на ошибках предыдущих.
Bias-Variance в Boosting

- **Bagging** (Random Forest) уменьшает **дисперсию** (variance) → борется с переобучением
- **Boosting** уменьшает **смещение** (bias) → улучшает подгонку к данным

Риск: Boosting может **переобучиться**, если слишком долго обучать.

17.2. Градиентный бустинг

Gradient Boosting — это метод бустинга, где каждая новая модель предсказывает **градиент функции потерь**.

Ключевая идея: оптимизация функции потерь через последовательное добавление моделей.

Формула итоговой модели:

$$F_T(x) = F_0(x) + \sum_{t=1}^{T} \eta \cdot h_t(x)$$

где:

- $F_0(x)$ — начальное предсказание (обычно среднее значение y)
- $h_t(x)$ — t-я базовая модель (обычно дерево решений)

- η — learning rate (скорость обучения)
- T — количество итераций (деревьев)

Алгоритм Gradient Boosting

Вход: данные $\{(x_i, y_i)\}_{i=1}^m$, функция потерь $L(y, F(x))$, количество итераций T, learning rate η

Алгоритм:

1. Инициализация: $F_0(x) = \arg\min_\gamma \sum_{i=1}^m L(y_i, \gamma)$ (например, $F_0 = \text{mean}(y)$)
2. Для $t = 1, 2, \ldots, T$:
   (a) Вычислить **псевдо-остатки** (градиент функции потерь):
   $$r_{it} = -\left[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}\right]_{F=F_{t-1}}$$
   (b) Обучить базовую модель $h_t(x)$ на данных $\{(x_i, r_{it})\}_{i=1}^m$
   (c) Найти оптимальный шаг γ_t:
   $$\gamma_t = \arg\min_\gamma \sum_{i=1}^m L(y_i, F_{t-1}(x_i) + \gamma \cdot h_t(x_i))$$
   (d) Обновить модель:
   $$F_t(x) = F_{t-1}(x) + \eta \cdot \gamma_t \cdot h_t(x)$$
3. **Выход**: $F_T(x)$

Почему "градиентный"

Псевдо-остатки r_{it} — это **антиградиент** функции потерь:

$$r_{it} = -\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}$$

Мы двигаемся в направлении антиградиента, чтобы минимизировать потери — как в градиентном спуске.

Примеры для разных функций потерь:

- **MSE** (регрессия): $L(y, F) = \frac{1}{2}(y - F)^2$
$$r_i = -\frac{\partial L}{\partial F} = -(F - y) = y - F(x_i) \quad \text{(остаток)}$$
- **Log-loss** (классификация): $L(y, F) = \log(1 + e^{-yF})$
$$r_i = \frac{y}{1 + e^{yF(x_i)}}$$

17.3. Пример: регрессия с MSE

Задача: предсказать y по x
Функция потерь: MSE = $\frac{1}{2}(y - F(x))^2$
Шаги:

1. $F_0(x) = \text{mean}(y)$ (начальное предсказание)
2. Вычислить остатки: $r_i = y_i - F_0(x_i)$ (ошибки)
3. Обучить дерево $h_1(x)$ на (x_i, r_i)
4. Обновить: $F_1(x) = F_0(x) + \eta \cdot h_1(x)$
5. Повторить шаги 2-4 для $t = 2, 3, \ldots, T$

Интерпретация: каждое новое дерево предсказывает **ошибку** предыдущей модели.

17.4. Деревья решений в бустинге

Градиентный бустинг обычно использует **деревья решений** как базовые модели.
Почему деревья:

- Могут аппроксимировать любую функцию
- Быстро обучаются
- Не требуют нормализации признаков
- Автоматически находят нелинейные зависимости
- Обрабатывают категориальные признаки

Мелкие деревья (weak learners)
В бустинге используют **мелкие деревья** (глубина 3-8):

- Низкое смещение: каждое дерево слабое, но много деревьев вместе сильные
- Быстрое обучение: мелкие деревья обучаются быстро
- Регуляризация: предотвращает переобучение

Глубина дерева (`max_depth`):

- 1-2: очень слабые деревья (пеньки)
- 3-6: оптимально для большинства задач
- 7-10: для сложных зависимостей
- > 10: риск переобучения

17.5. Гиперпараметры градиентного бустинга

Trade-off: $n\_estimators \times learning\_rate \approx const$

- Большой η (0.3) + мало деревьев (100) = быстро, но хуже качество
- Малый η (0.01) + много деревьев (1000) = медленно, но лучше качество

Параметр	Описание	Рекомендации
`n_estimators`	Количество деревьев	100-1000. Больше = лучше, но медленнее
`learning_rate` (η)	Скорость обучения	0.01-0.3. Меньше η = больше деревьев нужно
`max_depth`	Глубина деревьев	3-8. Мелкие деревья = меньше переобучение
`subsample`	Доля объектов для обучения	0.5-1.0. < 1 = стохастический градиентный бустинг
`colsample_bytree`	Доля признаков для дерева	0.3-1.0. Меньше = больше разнообразие
`min_child_weight`	Минимальная сумма весов в листе	1-10. Больше = меньше переобучение
`reg_alpha`	L1 регуляризация	0-1. Для разреженных моделей
`reg_lambda`	L2 регуляризация	0-10. Для сглаживания весов

Таблица 5: Основные гиперпараметры градиентного бустинга

17.6. scikit-learn: GradientBoostingClassifier

```python
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# Данные
data = load_breast_cancer()
X, y = data.data, data.target
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=42
)

# Gradient Boosting
gb = GradientBoostingClassifier(
    n_estimators=100,
    learning_rate=0.1,
    max_depth=3,
    random_state=42
)

gb.fit(X_train, y_train)

# Предсказание
y_pred = gb.predict(X_test)
print(f"Accuracy: {accuracy_score(y_test, y_pred):.4f}")
```

17.7. Staged predictions

Staged predictions — предсказания после каждого дерева. Полезно для анализа сходимости.

```python
import matplotlib.pyplot as plt
import numpy as np
```

```

# Staged predictions
train_scores = []
test_scores = []

for y_pred_train in gb.staged_predict(X_train):
    train_scores.append(accuracy_score(y_train, y_pred_train))

for y_pred_test in gb.staged_predict(X_test):
    test_scores.append(accuracy_score(y_test, y_pred_test))

# Визуализация
plt.figure(figsize=(10, 6))
plt.plot(train_scores, label='Train')
plt.plot(test_scores, label='Test')
plt.xlabel('Number of trees')
plt.ylabel('Accuracy')
plt.title('Gradient Boosting: staged predictions')
plt.legend()
plt.show()
```

17.8. XGBoost

XGBoost (eXtreme Gradient Boosting) — оптимизированная реализация градиентного бустинга.

Преимущества:

- Скорость: параллелизация, оптимизированный код на C++
- Регуляризация: L1 и L2 регуляризация весов листьев
- Обработка пропусков: автоматическая работа с missing values
- Pruning: обрезка деревьев после построения (вместо pre-pruning)
- Cross-validation: встроенная CV во время обучения
- Early stopping: автоматическая остановка при отсутствии улучшения

Установка

```
pip install xgboost
```

Базовое использование

```
import xgboost as xgb
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# Данные
data = load_breast_cancer()
X, y = data.data, data.target
X_train, X_test, y_train, y_test = train_test_split(
```

```
    X, y, test_size=0.3, random_state=42
)

# XGBoost Classifier
xgb_clf = xgb.XGBClassifier(
    n_estimators=100,
    learning_rate=0.1,
    max_depth=3,
    random_state=42,
    eval_metric='logloss'
)

xgb_clf.fit(X_train, y_train)
y_pred = xgb_clf.predict(X_test)

print(f"XGBoost Accuracy: {accuracy_score(y_test, y_pred):.4f}")
```

Early Stopping

```
# Early stopping
xgb_clf_es = xgb.XGBClassifier(
    n_estimators=1000,
    learning_rate=0.05,
    max_depth=3,
    random_state=42,
    early_stopping_rounds=10,
    eval_metric='logloss'
)

xgb_clf_es.fit(
    X_train, y_train,
    eval_set=[(X_test, y_test)],
    verbose=False
)

print(f"Best iteration: {xgb_clf_es.best_iteration}")
print(f"Best score: {xgb_clf_es.best_score:.4f}")
```

Feature Importance

```
import matplotlib.pyplot as plt

# Feature importance
importances = xgb_clf.feature_importances_
indices = np.argsort(importances)[::-1][:15]

plt.figure(figsize=(10, 6))
plt.bar(range(15), importances[indices])
plt.xticks(range(15),
           [data.feature_names[i] for i in indices],
           rotation=45, ha='right')
plt.title('XGBoost Feature Importances')
```

```
plt.tight_layout()
plt.show()
```

Основные параметры XGBoost

Параметр	Описание
n_estimators	Количество деревьев (100-1000)
learning_rate	Скорость обучения (0.01-0.3)
max_depth	Глубина деревьев (3-10)
subsample	Доля объектов (0.5-1.0)
colsample_bytree	Доля признаков для каждого дерева (0.3-1.0)
colsample_bylevel	Доля признаков для каждого уровня дерева
min_child_weight	Минимальная сумма весов в листе (1-10)
gamma	Минимальное уменьшение loss для split (0-5)
reg_alpha	L1 регуляризация (0-1)
reg_lambda	L2 регуляризация (1-10)
scale_pos_weight	Балансировка классов
early_stopping_rounds	Остановка при отсутствии улучшения

Таблица 6: Параметры XGBoost

17.9. LightGBM

LightGBM (Light Gradient Boosting Machine) — быстрая реализация градиентного бустинга от Microsoft.

Ключевые особенности:

- Leaf-wise рост: деревья растут по листьям, а не по уровням (быстрее сходимость)
- GOSS (Gradient-based One-Side Sampling): умное сэмплирование объектов
- EFB (Exclusive Feature Bundling): объединение признаков
- Скорость: в 2-10 раз быстрее XGBoost
- Точность: сравнимая или лучше XGBoost

Level-wise vs Leaf-wise
XGBoost (level-wise):

- Растёт по уровням: сначала все узлы уровня 1, потом уровня 2, и т.д.
- Симметричные деревья
- Медленнее сходимость

LightGBM (leaf-wise):

- Растёт по листьям: выбирает лист с максимальным приростом информации
- Несимметричные деревья
- Быстрее сходимость
- Риск переобучения (нужен контроль глубины)

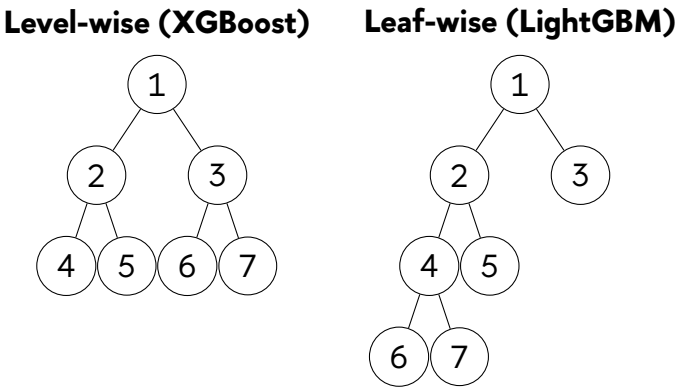


Рис. 20: Level-wise vs Leaf-wise рост деревьев

Установка

```
pip install lightgbm
```

Базовое использование

```
import lightgbm as lgb
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# Данные
data = load_breast_cancer()
X, y = data.data, data.target
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=42
)

# LightGBM Classifier
lgb_clf = lgb.LGBMClassifier(
    n_estimators=100,
    learning_rate=0.1,
    max_depth=3,
    num_leaves=31,  # должно быть < 2^max_depth
    random_state=42
)

lgb_clf.fit(X_train, y_train)
y_pred = lgb_clf.predict(X_test)

print(f"LightGBM Accuracy: {accuracy_score(y_test, y_pred):.4f}")
```

Основные параметры LightGBM

17.10. CatBoost

CatBoost (Categorical Boosting) — градиентный бустинг от Яндекса, специализирующийся на категориальных признаках.

Ключевые особенности:

Параметр	Описание
num_leaves	Максимум листьев в дереве (31). Должно быть $< 2^{\text{max\_depth}}$
max_depth	Глубина деревьев (-1 = без ограничений, рекомендуется 3-10)
learning_rate	Скорость обучения (0.01-0.3)
n_estimators	Количество деревьев (100-1000)
subsample	Доля объектов (0.5-1.0)
colsample_bytree	Доля признаков (0.3-1.0)
min_child_samples	Минимум объектов в листе (20)
reg_alpha	L1 регуляризация
reg_lambda	L2 регуляризация

Таблица 7: Параметры LightGBM

- Категориальные признаки: встроенная обработка без one-hot encoding
- Ordered boosting: защита от переобучения через упорядоченные предсказания
- Symmetric trees: симметричные деревья (oblivious trees)
- Ordered target encoding: для категорий
- GPU: отличная поддержка GPU

Ordered Target Encoding
Проблема: стандартный target encoding переобучается
Решение CatBoost: ordered target encoding

- Для каждого объекта i используем только объекты $j < i$ для вычисления статистики категории
- Предотвращает target leakage.

Symmetric Trees

Oblivious trees (забывчивые деревья) — симметричные деревья, где все узлы на одном уровне используют один и тот же признак для split.

Преимущества:

- Меньше переобучение
- Быстрые предсказания (можно использовать битовые операции)
- Меньше памяти

Установка

```
pip install catboost
```

Базовое использование

```
from catboost import CatBoostClassifier
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
```

```

# Данные
data = load_breast_cancer()
X, y = data.data, data.target
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=42
)

# CatBoost Classifier
cat_clf = CatBoostClassifier(
    iterations=100,
    learning_rate=0.1,
    depth=3,
    random_state=42,
    verbose=False
)

cat_clf.fit(X_train, y_train)
y_pred = cat_clf.predict(X_test)

print(f"CatBoost Accuracy: {accuracy_score(y_test, y_pred):.4f}")
```

Работа с категориальными признаками

```
import pandas as pd
from catboost import CatBoostClassifier

# Создаём данные с категориями
df = pd.DataFrame({
    'city': ['Moscow', 'SPb', 'Moscow', 'Kazan', 'SPb'],
    'district': ['Center', 'North', 'South', 'Center', 'Center'],
    'rooms': [2, 1, 3, 2, 1],
    'price': [100, 80, 120, 90, 85]
})

X = df[['city', 'district', 'rooms']]
y = df['price']

# Указываем категориальные признаки
cat_features = ['city', 'district']

# CatBoost автоматически обработает категории.
model = CatBoostClassifier(
    iterations=100,
    cat_features=cat_features,
    verbose=False
)

model.fit(X, y)
# Нет нужды в one-hot encoding или label encoding
```

Основные параметры CatBoost

Когда использовать:

Параметр	Описание
iterations	Количество деревьев (100-1000)
learning_rate	Скорость обучения (0.01-0.3)
depth	Глубина деревьев (4-10)
l2_leaf_reg	L2 регуляризация (1-10)
cat_features	Список индексов/имён категориальных признаков
border_count	Количество бинов для числовых признаков (32-255)
random_strength	Случайность при выборе split (0-10)
bagging_temperature	Температура для байесовского bagging (0-1)
task_type	'CPU' или 'GPU'

Таблица 8: Параметры CatBoost

- **sklearn GB**: простые задачи, быстрое прототипирование
- **XGBoost**: универсальный выбор, соревнования Kaggle
- **LightGBM**: большие данные ($> 10k$ примеров), скорость критична
- **CatBoost**: много категориальных признаков, мало времени на настройку

17.11. Практические рекомендации

Подбор гиперпараметров

1. Зафиксировать $n\_estimators = 100$, $learning\_rate = 0.1$
2. Подобрать `max_depth` (3-10)
3. Подобрать `subsample` и `colsample_bytree`
4. Подобрать регуляризацию (`reg_alpha`, `reg_lambda`)
5. Уменьшить `learning_rate` до 0.01-0.05
6. Увеличить `n_estimators` с `early_stopping`

Борьба с переобучением

- Уменьшить `max_depth`
- Увеличить `min_child_weight` / `min_child_samples`
- Уменьшить `subsample` и `colsample_bytree`
- Добавить регуляризацию (`reg_alpha`, `reg_lambda`)
- Использовать `early_stopping`

18. Задача кластеризации. Алгоритм Ллойда (K-Means)

18.1. Введение в кластеризацию

18.1.1 Что такое кластеризация

Определение 18.1. Кластеризация

Кластеризация — это задача **обучения без учителя** (unsupervised learning), где необходимо разбить множество объектов на группы (кластеры) таким образом, чтобы объекты внутри одной группы были похожи друг на друга, а объекты из разных групп — различались.

Отличие от классификации:

- **Классификация** (supervised learning): есть размеченные данные с метками классов
- **Кластеризация** (unsupervised learning): нет меток, алгоритм сам определяет группы

Постановка задачи: Дано множество объектов $X = \{x_1, x_2, \dots, x_m\}$, где каждый объект описывается признаками $x_i \in \mathbf{R}^n$. Нужно найти:

1. Число кластеров K
2. Функцию $c : X \to \{1, 2, \dots, K\}$, которая каждому объекту x_i присваивает номер кластера $c(x_i)$

18.1.2 Зачем нужна кластеризация

Применения:

- Сегментация клиентов: разбить покупателей на группы по поведению для таргетированной рекламы
- Сжатие данных: объединить похожие объекты для уменьшения размера данных
- Предобработка: выделить группы для дальнейшего анализа
- Поиск аномалий: объекты, не попавшие ни в один кластер, могут быть выбросами
- Анализ изображений: сегментация изображений, группировка похожих изображений
- Биоинформатика: группировка генов с похожим поведением
- Рекомендательные системы: группировка пользователей/товаров

Реальные примеры:

- Netflix группирует пользователей по предпочтениям для рекомендаций фильмов
- Банки выделяют группы клиентов с похожим финансовым поведением
- Супермаркеты анализируют покупательские корзины для оптимизации расположения товаров

18.1.3 Критерии качества кластеризации

1. Внутрикластерное расстояние (должно быть минимальным):

$$W = \sum_{k=1}^{K} \sum_{x_i \in C_k} \|x_i - \mu_k\|^2$$

где μ_k — центр кластера C_k, $\mu_k = \frac{1}{|C_k|} \sum_{x_i \in C_k} x_i$

2. Межкластерное расстояние (должно быть максимальным):

$$B = \sum_{k=1}^{K} |C_k| \cdot \|\mu_k - \mu\|^2$$

где μ — центр всех данных, $\mu = \frac{1}{m} \sum_{i=1}^{m} x_i$

3. Отношение (должно стремиться к минимуму):

$$Q = \frac{W}{B}$$

18.2. Алгоритм K-Means (Ллойда)

18.2.1 Идея алгоритма

K-Means (метод K-средних, алгоритм Ллойда) — один из самых простых и популярных алгоритмов кластеризации.

Главная идея: итеративно находить центры кластеров и присваивать каждый объект ближайшему центру.

Ключевые особенности:

- Число кластеров K задаётся заранее
- Использует евклидову метрику (обычно)
- Итеративный алгоритм
- Всегда сходится к некоторому локальному минимуму

18.2.2 Алгоритм K-Means

Вход:

- Данные $X = \{x_1, x_2, \dots, x_m\}$, где $x_i \in \mathbf{R}^n$
- Число кластеров K
- Метрика расстояния $d(x, y)$ (обычно евклидово расстояние)

Алгоритм:

1. **Инициализация**: случайно выбрать K центров кластеров $\mu_1, \mu_2, \dots, \mu_K$
   - Вариант 1: случайно выбрать K объектов из данных
   - Вариант 2: случайно сгенерировать K точек в признаковом пространстве

2. **Назначение кластеров**: для каждого объекта x_i найти ближайший центр:

$$c(x_i) = \arg\min_{k=1,\dots,K} d(x_i, \mu_k)$$

3. **Пересчёт центров**: для каждого кластера k пересчитать центр как среднее всех объектов в кластере:

$$\mu_k = \frac{1}{|C_k|} \sum_{x_i \in C_k} x_i$$

4. **Проверка сходимости**: если центры не изменились (или изменились незначительно), то **стоп**, иначе перейти к шагу 2

Выход: метки кластеров $c(x_i)$ для каждого объекта x_i

18.2.3 Математическое обоснование

K-Means минимизирует **внутрикластерную дисперсию**:

$$J = \sum_{k=1}^{K} \sum_{x_i \in C_k} \|x_i - \mu_k\|^2$$

Теорема: Алгоритм K-Means всегда сходится к некоторому локальному минимуму функции J.

Доказательство (интуиция):

- На шаге назначения кластеров мы уменьшаем J (присваиваем объекты ближайшим центрам)
- На шаге пересчёта центров мы также уменьшаем J (центр — это оптимальная точка для минимизации суммы квадратов расстояний)
- Функция $J \geq 0$ и ограничена снизу
- Следовательно, алгоритм сходится

Важно: K-Means сходится к **локальному** минимуму, который зависит от начальной инициализации!

18.2.4 Метрики расстояния

Евклидово расстояние (по умолчанию):

$$d(x, y) = \|x - y\|_2 = \sqrt{\sum_{j=1}^{n} (x_j - y_j)^2}$$

Манхэттенское расстояние:

$$d(x, y) = \|x - y\|_1 = \sum_{j=1}^{n} |x_j - y_j|$$

Косинусное расстояние:

$$d(x, y) = 1 - \frac{x \cdot y}{\|x\| \cdot \|y\|}$$

18.3. Инициализация центров: K-Means++

18.3.1 Проблема случайной инициализации

Проблема: случайная инициализация центров может привести к плохим результатам.
Пример: если два центра инициализированы близко друг к другу, они могут остаться в одном кластере.

18.3.2 K-Means++ алгоритм

Идея: инициализировать центры так, чтобы они были далеко друг от друга.
Алгоритм:

1. Случайно выбрать первый центр μ_1 из данных
2. Для каждого объекта x_i вычислить расстояние до ближайшего уже выбранного центра:

$$D(x_i) = \min_{j=1,\dots,k} d(x_i, \mu_j)$$

3. Выбрать следующий центр μ_{k+1} с вероятностью, пропорциональной $D(x_i)^2$:

$$P(x_i) = \frac{D(x_i)^2}{\sum_{j=1}^{m} D(x_j)^2}$$

4. Повторить шаги 2-3, пока не выбраны все K центров

Преимущество: K-Means++ гарантирует лучшую инициализацию и обычно быстрее сходится.

18.4. Выбор числа кластеров K

18.4.1 Метод локтя (Elbow Method)

Идея: построить график зависимости внутрикластерной дисперсии J от числа кластеров K.

$$J(K) = \sum_{k=1}^{K} \sum_{x_i \in C_k} \|x_i - \mu_k\|^2$$

Алгоритм:

1. Запустить K-Means для разных K (например, $K = 1, 2, \dots, 10$)
2. Вычислить $J(K)$ для каждого K
3. Построить график $J(K)$ от K
4. Найти “локоть” — точку, после которой $J(K)$ уменьшается медленно

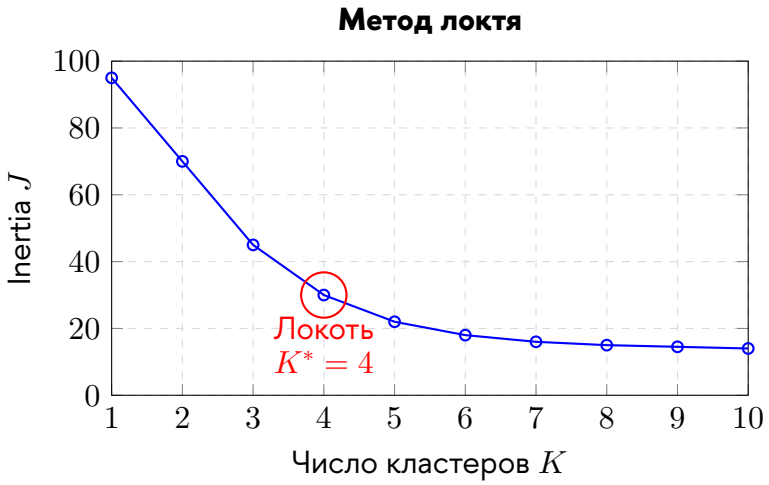


Рис. 21: Метод локтя для выбора K

18.4.2 Silhouette Score

Silhouette коэффициент измеряет, насколько объект похож на свой кластер по сравнению с другими кластерами.

Для объекта $x_i \in C_k$:

1. a_i — среднее расстояние до всех объектов в своём кластере:

$$a_i = \frac{1}{|C_k| - 1} \sum_{x_j \in C_k, j \neq i} d(x_i, x_j)$$

2. b_i — минимальное среднее расстояние до объектов в ближайшем другом кластере:

$$b_i = \min_{l \neq k} \frac{1}{|C_l|} \sum_{x_j \in C_l} d(x_i, x_j)$$

3. Silhouette коэффициент:

$$s_i = \frac{b_i - a_i}{\max(a_i, b_i)}$$

Интерпретация:

- $s_i \approx 1$: объект хорошо вписывается в свой кластер
- $s_i \approx 0$: объект на границе между кластерами
- $s_i \approx -1$: объект, вероятно, отнесён к неправильному кластеру

Средний Silhouette Score:

$$\bar{s} = \frac{1}{m} \sum_{i=1}^{m} s_i$$

Выбираем K с максимальным $\bar{s}$.

18.5. Преимущества и недостатки K-Means

Преимущества

- Простота: алгоритм очень простой и понятный
- Скорость: быстро работает даже на больших данных ($O(nKmI)$, где I — число итераций)
- Масштабируемость: легко распараллелить
- Сходимость: всегда сходится (к локальному минимуму)
- Интерпретируемость: легко понять и объяснить результаты

Недостатки

- Нужно задавать K: число кластеров нужно знать заранее
- Чувствительность к инициализации: разные начальные центры дают разные результаты
- Только сферические кластеры: плохо работает с кластерами произвольной формы
- Чувствительность к выбросам: выбросы сильно влияют на центры
- Одинаковый размер: предполагает, что кластеры примерно одинакового размера
- Евклидова метрика: предполагает изотропную дисперсию (одинаковую во всех направлениях)

18.6. Реализация в scikit-learn

Базовое использование

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

# Данные
X = np.array([[1, 2], [1.5, 1.8], [5, 8], [8, 8], [1, 0.6], [9, 11]])

# K-Means с 2 кластерами
kmeans = KMeans(n_clusters=2, random_state=42)
kmeans.fit(X)

# Метки кластеров
labels = kmeans.labels_
print("Метки:", labels)  # [0 0 1 1 0 1]

# Центры кластеров
centers = kmeans.cluster_centers_
print("Центры:", centers)

# Предсказание для новых данных
new_point = np.array([[3, 4]])
prediction = kmeans.predict(new_point)
print("Кластер для [3, 4]:", prediction)
```

Параметр	Описание
`n_clusters`	Число кластеров K (по умолчанию 8)
`init`	Метод инициализации: `'k-means++'` (по умолчанию), `'random'`
`n_init`	Число запусков алгоритма с разными начальными центрами (по умолчанию 10)
`max_iter`	Максимальное число итераций (по умолчанию 300)
`tol`	Критерий остановки: если изменение центров $< tol$, то стоп (по умолчанию 1e-4)
`random_state`	Seed для воспроизводимости
`algorithm`	`'lloyd'` (стандартный), `'elkan'` (быстрее для плотных данных), `'auto'`

Таблица 9: Параметры K-Means в sklearn

Основные параметры
Атрибуты после обучения

```
# Метки кластеров для обучающих данных
labels = kmeans.labels_

# Центры кластеров
centers = kmeans.cluster_centers_

# Внутрикластерная дисперсия (inertia)
inertia = kmeans.inertia_

# Число итераций до сходимости
n_iter = kmeans.n_iter_
```

19. Алгоритм DBSCAN

19.1. Введение в DBSCAN

Вспомним основные **недостатки K-Means**:

- Нужно заранее задавать число кластеров K
- Работает только со сферическими (гауссовскими) кластерами
- Чувствителен к выбросам и шуму
- Предполагает, что все объекты принадлежат какому-то кластеру

DBSCAN решает эти проблемы.

Определение 19.1. DBSCAN

DBSCAN (Density-Based Spatial Clustering of Applications with Noise) — алгоритм кластеризации, основанный на плотности, который:

- Автоматически определяет число кластеров
- Находит кластеры произвольной формы
- Выделяет шумовые выбросы
- Не требует задавать K

Почему популярен до сих пор?

- Алгоритмический подход (без вероятностей)
- Эффективные эвристики
- Хорошо работает на реальных данных
- Находит кластеры произвольной формы
- Выделяет аномалии (выбросы, шум)

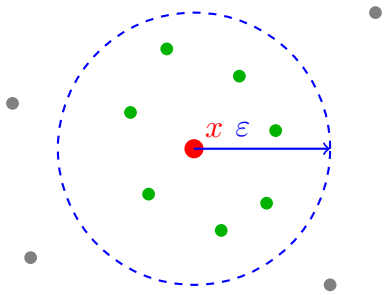
19.2. Основные понятия DBSCAN

Определение 19.2. ε-окрестность

Для объекта $x \in X$ и параметра $\varepsilon > 0$, **ε-окрестность** $N_\varepsilon(x)$ — это множество всех точек, находящихся на расстоянии не более ε от x:

$$N_\varepsilon(x) = \{x' \in X \mid d(x, x') \leq \varepsilon\}$$

где $d(x, x')$ — метрика расстояния (обычно евклидова).



$N_\varepsilon(x)$ — окрестность радиуса ε
Зелёные точки $\in N_\varepsilon(x)$

Рис. 22: ε-окрестность точки x

19.2.1 Типы точек

DBSCAN классифицирует все точки на **три типа**:

Определение 19.3. Типы точек в DBSCAN

Пусть $\varepsilon > 0$ и $m \in \mathcal{N}$ — параметры алгоритма.

1. **Корневая точка** (core point): если $|N_\varepsilon(x)| \geq m$
   - В ε-окрестности находится не менее m точек (включая саму x)
   - Это "плотные" области данных
2. **Граничная точка** (border point): если $|N_\varepsilon(x)| < m$, но x находится в ε-окрестности какой-то корневой точки
   - Недостаточно соседей, чтобы быть корневой
   - Но достаточно близко к корневой точке
3. **Шумовая точка** (noise point): если $|N_\varepsilon(x)| < m$ и x не находится в ε-окрестности ни одной корневой точки
   - Выброс, аномалия
   - Не принадлежит ни одному кластеру

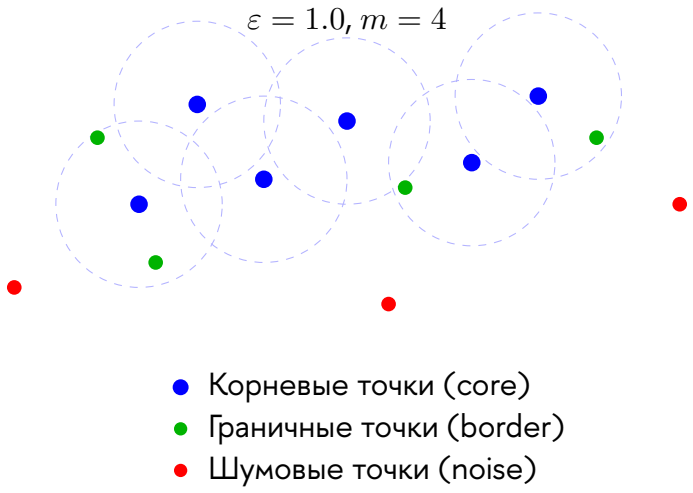


Рис. 23: Типы точек в DBSCAN

Алгоритм DBSCAN
Вход:

- Выборка $X = \{x_1, \dots, x_n\}$
- Параметр ε (радиус окрестности)
- Параметр m (минимальное число соседей, MinPts)

Выход: разбиение выборки на кластеры и шумовые выбросы N

Алгоритм:

1. Инициализация: $U = X$ — непомеченные точки, $N = \emptyset$, $a = 0$ (номер кластера)
2. Пока в выборке есть непомеченные точки ($U \neq \emptyset$):
   (a) Выбирается случайный образ $x \in U$
   (b) Вычисляется окрестность: $U_\varepsilon(x) = \{x' \in X : \rho(x, x') < \varepsilon\}$
   (c) Если $|U_\varepsilon(x)| < m$ (точка x — потенциальный шум):
      - Образ x помечается как возможный шумовой выброс
   (d) Иначе (точка x — корневая, начало нового кластера):
      i. Создаётся новый кластер: $K = U_\varepsilon(x)$, $a = a + 1$
      ii. Для всех $x' \in K$, не помеченных или шумовых:
         A. Вычисляется окрестность: $U_\varepsilon(x')$
         B. Если $|U_\varepsilon(x')| \geq m$ (точка x' — корневая):
            - Расширяем кластер: $K = K \cup U_\varepsilon(x')$
         C. Иначе (точка x' — граничная):
            - Помечаем x' как граничную точку кластера K
      iii. Присваиваем всем $x_i \in K$ метку кластера: $a_i = a$
      iv. Удаляем обработанные точки: $U = U \setminus K$
3. Все оставшиеся точки без метки кластера помечаются как шумовые выбросы N

Результат: каждой точке $x_i \in X$ присвоена метка кластера $a_i \in \{1, 2, \dots, a\}$ или метка шума ($a_i = -1$)

19.3. Параметры DBSCAN

19.3.1 Параметр ε (eps)

ε — радиус окрестности для поиска соседей.
Как выбрать?
Метод k-distance graph:

1. Для каждой точки x_i найти расстояние до k-го ближайшего соседа (обычно $k = m$)
2. Отсортировать эти расстояния по возрастанию
3. Построить график
4. Найти "локоть" — резкий скачок расстояний
5. Значение в "локте" — хороший ε

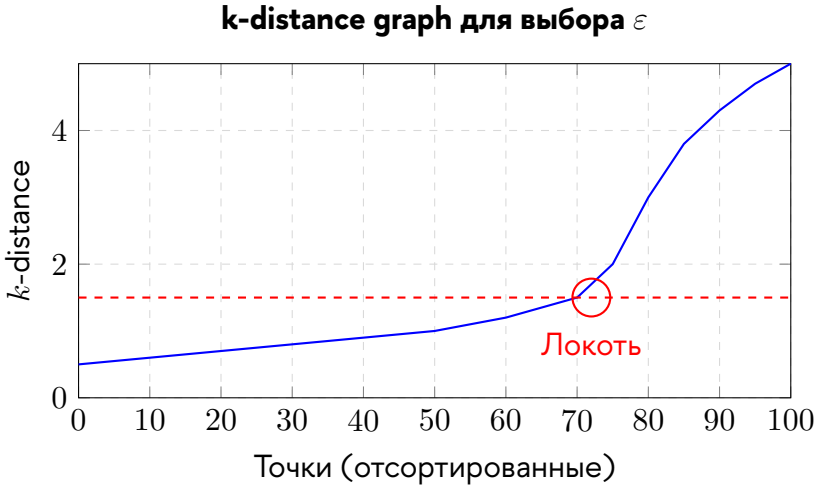


Рис. 24: k-distance graph для выбора ε

Влияние ε:

- **Слишком маленький** ε: много маленьких кластеров и шума
- **Слишком большой** ε: все точки объединяются в один кластер

19.3.2 Параметр m (MinPts, min_samples)

m (MinPts) — минимальное число точек в ε-окрестности для корневой точки.
Как выбрать?
Эмпирическое правило:

$$m = 2 \times \dim(X)$$

где $\dim(X)$ — размерность данных.

Примеры:

- 2D данные: $m = 4$
- 3D данные: $m = 6$
- Высокоразмерные данные: $m = 2 \times n$, где n — число признаков

Влияние m:

- **Маленький** m: больше кластеров, менее устойчив к шуму
- **Большой** m: меньше кластеров, более устойчив к шуму, но может пропустить мелкие кластеры

19.4. Преимущества и недостатки DBSCAN

Преимущества:

- Не нужно задавать число кластеров — определяется автоматически
- Находит кластеры произвольной формы — не только сферические
- Устойчив к выбросам — выделяет шум отдельно
- Работает с любой метрикой — евклидова, манхэттен, косинусная и т.д.
- Детерминированный — всегда один и тот же результат (в отличие от K-Means)
- Эффективен — $O(m \log m)$ с пространственными индексами

Недостатки:

- Нужно подбирать параметры ε и m — может быть непросто
- Плохо работает с кластерами разной плотности — ε фиксирован для всех кластеров
- Чувствителен к выбору ε — небольшое изменение может сильно повлиять на результат
- Проблемы в высоких размерностях — "проклятие размерности", все точки далеко друг от друга
- Граничные точки могут быть отнесены к разным кластерам

Критерий	**K-Means**	**DBSCAN**
Число кластеров K	Нужно задавать	Автоматически
Форма кластеров	Только сферические	Произвольная
Выбросы	Нет (все в кластере)	Да (шумовые точки)
Чувствительность к параметрам	Сильная (к K)	Средняя (к ε, m)
Детерминированность	Нет (зависит от init)	Да
Кластеры разной плотности	Плохо	Плохо
Интерпретируемость	Высокая	Средняя

Таблица 10: Сравнение K-Means и DBSCAN

19.5. Реализация в scikit-learn

19.5.1 Базовое использование

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

# Данные
X = np.array([[1, 2], [2, 2], [2, 3], [8, 7], [8, 8], [25, 80]])

# DBSCAN
dbscan = DBSCAN(eps=3, min_samples=2)
labels = dbscan.fit_predict(X)

print("Метки:", labels)  # [ 0  0  0  1  1 -1]
# -1 означает шум

# Число кластеров (исключая шум)
n_clusters = len(set(labels)) - (1 if -1 in labels else 0)
print(f"Число кластеров: {n_clusters}")

# Число шумовых точек
n_noise = list(labels).count(-1)
print(f"Число шумовых точек: {n_noise}")
```

19.5.2 Параметры

Параметр	Описание
`eps`	Радиус ε-окрестности (по умолчанию 0.5)
`min_samples`	Минимальное число точек в окрестности для корневой точки (по умолчанию 5)
`metric`	Метрика расстояния: `'euclidean'` (по умолчанию), `'manhattan'`, `'cosine'` и др.
`algorithm`	Алгоритм поиска соседей: `'auto'`, `'ball_tree'`, `'kd_tree'`, `'brute'`
`leaf_size`	Размер листа для BallTree/KDTree (по умолчанию 30)
`n_jobs`	Число потоков для параллельных вычислений (`-1` = все ядра)

Таблица 11: Параметры DBSCAN в sklearn

19.5.3 Атрибуты после обучения

```python
# После dbscan.fit(X)

# Метки кластеров (-1 = шум)
labels = dbscan.labels_

# Индексы корневых точек
core_sample_indices = dbscan.core_sample_indices_

```

```
# Маска корневых точек (булев массив)
core_samples_mask = np.zeros_like(labels, dtype=bool)
core_samples_mask[core_sample_indices] = True

# Компоненты (кластеры)
components = dbscan.components_  # Только корневые точки
```

20. Метод опорных векторов (SVM)

20.1. Введение

Метод опорных векторов (англ. *Support Vector Machine*, SVM) — это мощный алгоритм машинного обучения с учителем, применяемый для задач классификации и регрессии. SVM принадлежит к семейству линейных классификаторов и основан на поиске оптимальной разделяющей гиперплоскости между классами.

Ключевые идеи SVM

- Максимизация зазора (margin) — SVM ищет гиперплоскость, которая максимально удалена от ближайших точек обоих классов
- Опорные векторы — только ближайшие к границе точки (опорные векторы) определяют положение разделяющей гиперплоскости
- Ядровой трюк — позволяет решать нелинейные задачи, неявно отображая данные в пространство более высокой размерности
- Устойчивость к переобучению — благодаря принципу максимизации зазора SVM обладает хорошей обобщающей способностью

Когда применять SVM?

- Данные линейно или почти линейно разделимы
- Число признаков велико по сравнению с числом объектов
- Задачи с высокой размерностью (текстовая классификация, биоинформатика)
- Необходима высокая точность классификации
- Есть выбросы в данных (SVM устойчив к ним)

20.2. Линейный SVM: математическая формулировка

20.2.1 Разделяющая гиперплоскость

Пусть дана обучающая выборка $\{(x_i, y_i)\}_{i=1}^n$, где $x_i \in \mathbb{R}^d$ — вектор признаков, $y_i \in \{-1, +1\}$ — метка класса.

Разделяющая гиперплоскость задаётся уравнением:

$$w^T x + b = 0$$

где:

- $w \in \mathbb{R}^d$ — вектор нормали к гиперплоскости (веса)
- $b \in \mathbb{R}$ — смещение (bias)
- $w^T x$ — скалярное произведение

Решающее правило классификации:

$$\hat{y} = \text{sign}(w^T x + b) = \begin{cases} +1, & \text{если } w^T x + b > 0 \\ -1, & \text{если } w^T x + b < 0 \end{cases}$$

20.2.2 Зазор (Margin)

Расстояние от точки x до гиперплоскости:

$$\rho(x, (w, b)) = \frac{|w^T x + b|}{\|w\|}$$

Зазор (margin) — это удвоенное минимальное расстояние от точек до гиперплоскости:

$$\gamma = 2 \cdot \min_{i=1,\dots,n} \frac{|w^T x_i + b|}{\|w\|}$$

Для корректно классифицированных точек выполняется:

$$y_i(w^T x_i + b) > 0, \quad \forall i = 1, \dots, n$$

20.2.3 Опорные векторы

Опорные векторы (support vectors) — это точки обучающей выборки, которые лежат на границах зазора (margin):

$$y_i(w^T x_i + b) = 1$$

Именно эти точки определяют положение оптимальной разделяющей гиперплоскости. Удаление других точек не изменит решение.

20.2.4 Задача оптимизации: жёсткий margin

Цель: найти гиперплоскость с максимальным зазором.
Максимизация зазора эквивалентна минимизации нормы весов:

$$\begin{cases} \frac{1}{2}\|w\|^2 \to \min_{w,b} \\ y_i(w^T x_i + b) \geq 1, \quad \forall i = 1, \dots, n \end{cases}$$

Почему минимизируем $\|w\|^2$?
Зазор $\gamma = \frac{2}{\|w\|}$, поэтому максимизация $\gamma \Leftrightarrow$ минимизация $\|w\|$.

20.2.5 Soft Margin: мягкий зазор

В реальных данных часто присутствуют выбросы и классы не являются линейно разделимыми. Для этого вводят **slack-переменные** $\xi_i \geq 0$:

$$\begin{cases} \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{n} \xi_i \to \min_{w,b,\xi} \\ y_i(w^T x_i + b) \geq 1 - \xi_i, \quad \forall i \\ \xi_i \geq 0, \quad \forall i \end{cases}$$

где:

- ξ_i — величина нарушения ограничения для i-го объекта
- $C > 0$ — параметр регуляризации (гиперпараметр)

Интерпретация параметра C:

- Большой C — строгая классификация, узкий margin, риск переобучения
- Малый C — широкий margin, больше ошибок на обучении, лучшая обобщающая способность

20.3. Нелинейный SVM: ядровой трюк

20.3.1 Проблема линейной неразделимости

Многие реальные данные не являются линейно разделимыми в исходном пространстве признаков. Например, данные в форме концентрических окружностей или полумесяцев.

Идея: отобразить данные в пространство более высокой размерности, где они станут линейно разделимыми.

20.3.2 Отображение в пространство признаков

Пусть $\varphi : \mathbb{R}^d \to \mathbb{R}^D$ — нелинейное отображение в пространство признаков, где $D \gg d$ (возможно, $D = \infty$).

После отображения задача SVM:

$$\begin{cases} \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{n} \xi_i \to \min \\ y_i(w^T\varphi(x_i) + b) \geq 1 - \xi_i \end{cases}$$

Проблема: явное вычисление $\varphi(x)$ может быть вычислительно невозможно.

20.3.3 Ядровой трюк (Kernel Trick)

В двойственной задаче SVM нужны только скалярные произведения $\varphi(x_i)^T\varphi(x_j)$.

Ядровая функция $K(x, x')$ вычисляет скалярное произведение в пространстве признаков:

$$K(x, x') = \varphi(x)^T\varphi(x')$$

Преимущество: можно вычислить $K(x, x')$ напрямую, не вычисляя $\varphi(x)$.

20.3.4 Популярные ядра

1. **Линейное ядро**:
$$K(x, x') = x^T x'$$
Используется для линейно разделимых данных.

2. **Полиномиальное ядро**:
$$K(x, x') = (x^T x' + c)^d$$
где d — степень полинома, $c \geq 0$ — свободный член.

3. **RBF (Радиальная базисная функция) / Гауссово ядро**:
$$K(x, x') = \exp\left(-\gamma\|x - x'\|^2\right)$$
где $\gamma = \frac{1}{2\sigma^2}$ — параметр ширины ядра.

   Самое популярное ядро. Соответствует бесконечномерному пространству признаков.

4. **Сигмоидальное ядро**:
$$K(x, x') = \tanh(\gamma x^T x' + c)$$
Похоже на активационную функцию нейронных сетей.

20.3.5 Влияние параметров RBF-ядра

Параметр γ контролирует «радиус влияния» опорных векторов:

- Малый γ — широкий радиус, гладкая граница решения, риск недообучения
- Большой γ — узкий радиус, сложная граница, риск переобучения

Начинайте с $\gamma = \frac{1}{n \cdot \text{var}(X)}$, где n — число признаков.

20.4. Многоклассовая классификация с SVM

SVM изначально — бинарный классификатор. Для задач с $K > 2$ классами применяют стратегии декомпозиции.

20.4.1 One-vs-Rest (OvR) / One-vs-All (OvA)

Идея: обучить K бинарных классификаторов, где каждый отделяет один класс от всех остальных.

Алгоритм:

- Для каждого класса $k = 1, \dots, K$ обучить SVM, который отличает класс k от остальных классов
- Для нового объекта x: вычислить $f_k(x) = w_k^T x + b_k$ для всех k
- Выбрать класс с максимальным значением: $\hat{y} = \arg\max_k f_k(x)$

Преимущества:

- Простота реализации
- Число классификаторов: K (линейно растёт)

Недостатки:

- Дисбаланс классов (один против всех)
- Неоднозначность при близких значениях $f_k(x)$

20.4.2 One-vs-One (OvO)

Идея: обучить классификатор для каждой пары классов.

Алгоритм:

- Обучить $\frac{K(K-1)}{2}$ бинарных SVM для всех пар классов (i, j)
- Для нового объекта: каждый классификатор голосует за один из двух классов
- Выбрать класс с наибольшим числом голосов

Преимущества:

- Каждый классификатор обучается на меньшем подмножестве данных
- Нет проблемы дисбаланса классов

Недостатки:

- Число классификаторов: $O(K^2)$ (квадратично растёт)
- Более медленное предсказание

20.5. Практическая реализация с Scikit-learn

20.5.1 Линейный SVM

```
from sklearn.svm import LinearSVC
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline

# Pipeline: нормализация + SVM
model = Pipeline([
    ('scaler', StandardScaler()),
    ('svc', LinearSVC(C=1.0, max_iter=10000))
])

# Обучение
model.fit(X_train, y_train)

# Предсказание
y_pred = model.predict(X_test)
```

Важно: всегда нормализуйте данные перед SVM.

20.5.2 SVM с RBF-ядром

```
from sklearn.svm import SVC

# SVM с RBF-ядром
model = Pipeline([
    ('scaler', StandardScaler()),
    ('svc', SVC(kernel='rbf', C=1.0, gamma='scale'))
])

model.fit(X_train, y_train)
y_pred = model.predict(X_test)

# Вероятности классов
y_proba = model.predict_proba(X_test)
```

Параметры:

- `kernel`: 'linear', 'rbf', 'poly', 'sigmoid'
- C: параметр регуляризации
- `gamma`: параметр RBF-ядра ('scale' или 'auto')

20.5.3 Подбор гиперпараметров

```
from sklearn.model_selection import GridSearchCV

# Сетка параметров
param_grid = {
    'svc__C': [0.1, 1, 10, 100],
```

```
    'svc__gamma': [0.001, 0.01, 0.1, 1],
    'svc__kernel': ['rbf', 'poly']
}

# Grid Search
grid = GridSearchCV(
    model, param_grid,
    cv=5, scoring='f1', n_jobs=-1
)

grid.fit(X_train, y_train)

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

20.5.4 Многоклассовая классификация

```
from sklearn.datasets import load_iris
from sklearn.multiclass import OneVsRestClassifier,
 ↪  OneVsOneClassifier

# Загрузка данных Iris (3 класса)
iris = load_iris()
X, y = iris.data, iris.target

# One-vs-Rest
ovr = OneVsRestClassifier(SVC(kernel='rbf'))
ovr.fit(X_train, y_train)
y_pred_ovr = ovr.predict(X_test)

# One-vs-One
ovo = OneVsOneClassifier(SVC(kernel='rbf'))
ovo.fit(X_train, y_train)
y_pred_ovo = ovo.predict(X_test)
```

20.5.5 Оценка качества

```
from sklearn.metrics import (
    accuracy_score, precision_score, recall_score,
    f1_score, roc_auc_score, classification_report,
    confusion_matrix, ConfusionMatrixDisplay
)

# Метрики
print("Accuracy:", accuracy_score(y_test, y_pred))
print("Precision:", precision_score(y_test, y_pred))
print("Recall:", recall_score(y_test, y_pred))
print("F1:", f1_score(y_test, y_pred))

# ROC-AUC (для бинарной классификации)
y_proba = model.predict_proba(X_test)[:, 1]
print("ROC-AUC:", roc_auc_score(y_test, y_proba))
```

```

# Матрица ошибок
cm = confusion_matrix(y_test, y_pred)
disp = ConfusionMatrixDisplay(cm)
disp.plot()

# Подробный отчёт
print(classification_report(y_test, y_pred))
```

20.5.6 Визуализация границ решения

```
import numpy as np
import matplotlib.pyplot as plt

def plot_decision_boundary(model, X, y):
    h = 0.02  # шаг сетки
    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1

    xx, yy = np.meshgrid(
        np.arange(x_min, x_max, h),
        np.arange(y_min, y_max, h)
    )

    Z = model.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)

    plt.contourf(xx, yy, Z, alpha=0.3, cmap='viridis')
    plt.scatter(X[:, 0], X[:, 1], c=y,
                edgecolors='k', cmap='viridis')
    plt.xlabel('Feature 1')
    plt.ylabel('Feature 2')
    plt.title('SVM Decision Boundary')
    plt.show()

# Использование
plot_decision_boundary(model, X_test, y_test)
```

20.6. Преимущества и недостатки SVM

20.6.1 Преимущества

- Эффективен в высоких размерностях — когда число признаков $\gg$ число объектов
- Устойчив к переобучению — благодаря максимизации margin и регуляризации
- Работает с нелинейными данными — через ядровой трюк
- Универсальность — различные ядра позволяют адаптироваться к разным типам данных
- Устойчив к выбросам — опорные векторы определяют решение, остальные точки не важны
- Детерминированный — в отличие от нейронных сетей, всегда даёт один результат

20.6.2 Недостатки

- Медленное обучение — сложность $O(n^2 \cdot d)$ до $O(n^3 \cdot d)$ для больших данных
- Выбор ядра и параметров — требует кросс-валидации и экспериментов
- Не выдаёт вероятности напрямую — требуется дополнительная калибровка
- Чувствителен к масштабу признаков — обязательна нормализация
- Неинтерпретируем — особенно с нелинейными ядрами
- Не подходит для больших данных — $n > 100,000$ объектов

20.7. Рекомендации по применению

20.7.1 Выбор между Linear SVM и Kernel SVM

Используйте LinearSVC, если:

- Данные линейно разделимы
- Много признаков ($d > n$)
- Очень большая выборка ($n > 100,000$)
- Нужна скорость

Используйте SVC с RBF-ядром, если:

- Данные нелинейно разделимы
- Выборка небольшая или средняя ($n < 10,000$)
- Есть время на подбор гиперпараметров

20.7.2 Подбор гиперпараметров

Начальные значения:

- $C = 1.0$
- $\gamma = \frac{1}{n \cdot \text{var}(X)}$ (для RBF)

Поиск по сетке:

- $C \in \{0.1, 1, 10, 100\}$
- $\gamma \in \{0.001, 0.01, 0.1, 1\}$

21. Метод главных компонент (PCA)

21.1. Введение

Метод главных компонент (англ. *Principal Component Analysis*, PCA) — это метод линейного преобразования данных для понижения размерности путём проецирования на новое координатное пространство, оси которого (главные компоненты) направлены вдоль направлений максимальной дисперсии данных.

Ключевые идеи PCA

- Максимизация дисперсии — PCA ищет направления, вдоль которых данные имеют наибольшую вариабельность
- Понижение размерности — переход от d исходных признаков к k главным компонентам, где $k \ll d$
- Главные компоненты взаимно перпендикулярны и некоррелированы
- PCA не использует метки классов, работает только с признаками
- Линейное преобразование — каждая главная компонента есть линейная комбинация исходных признаков

Когда применять PCA?

- Визуализация многомерных данных в 2D или 3D пространстве
- Высокая размерность признаков (50+ признаков)
- Мультиколлинеарность между признаками
- Предобработка данных перед обучением моделей
- Шумоподавление и сжатие данных
- Ускорение обучения алгоритмов машинного обучения

21.2. Математические основы PCA

21.2.1 Постановка задачи

Дано:

- Матрица данных $X \in \mathbb{R}^{n \times d}$, где n — число объектов, d — число признаков
- Каждая строка $x_i \in \mathbb{R}^d$ — вектор признаков одного объекта

Цель:

- Найти $k < d$ новых признаков (главных компонент), максимально сохраняющих информацию
- Получить проекцию $Y \in \mathbb{R}^{n \times k}$ исходных данных

21.2.2 Центрирование данных

Первый шаг — вычесть среднее значение по каждому признаку:

$$\bar{x}_j = \frac{1}{n}\sum_{i=1}^{n} x_{ij}$$

Центрирование выполняется для каждого признака отдельно. Это переносит начало координат в центр данных.

Центрированная матрица:

$$X_{\text{centered}} = X - 1_n \bar{x}^T$$

где 1_n — вектор-столбец из единиц размера n.

Пример: Если исходная матрица $X = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}$, то средние $\bar{x}_1 = 3, \bar{x}_2 = 4$, и центрированная матрица $X_{\text{centered}} = \begin{bmatrix} -2 & -2 \\ 0 & 0 \\ 2 & 2 \end{bmatrix}$.

21.2.3 Стандартизация данных

Для данных с разными масштабами (например, вес в кг и рост в см) необходима стандартизация:

$$x_{ij}^{\text{std}} = \frac{x_{ij} - \bar{x}_j}{\sigma_j}$$

где $\sigma_j = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x_{ij} - \bar{x}_j)^2}$ — стандартное отклонение признака j.

Стандартизация приводит каждый признак к масштабу со средним 0 и стандартным отклонением 1. Это гарантирует, что признаки с разными единицами измерения будут вносить равный вклад в анализ.

Важно: Стандартизация критична для PCA. Без неё признаки с большим масштабом будут доминировать в главных компонентах.

Когда стандартизировать:

- Признаки в разных единицах (кг, см, годы)
- Признаки с сильно различающимися дисперсиями
- По умолчанию — всегда стандартизировать

Когда можно не стандартизировать:

- Все признаки в одних единицах (например, все — пиксели изображения)
- Хотите сохранить исходные масштабы признаков

21.2.4 Ковариационная матрица

Ковариационная матрица $\Sigma \in \mathbb{R}^{d \times d}$ — это квадратная симметричная матрица, которая характеризует взаимосвязь между всеми парами признаков.

Формула:

$$\Sigma = \frac{1}{n-1} X_{\text{centered}}^T X_{\text{centered}}$$

Разберём эту формулу детально:

1. X_{centered}^T — транспонированная центрированная матрица размера $d \times n$
2. $X_{\text{centered}}^T X_{\text{centered}}$ — матричное произведение размера $d \times d$
3. Деление на $(n-1)$ — несмещённая оценка ковариации (поправка Бесселя)

Элемент (j, k) ковариационной матрицы вычисляется как:

$$\sigma_{jk} = \frac{1}{n-1} \sum_{i=1}^{n} (x_{ij} - \bar{x}_j)(x_{ik} - \bar{x}_k)$$

Интерпретация элементов:

- При $j = k$: σ_{jj} — дисперсия признака j (мера разброса)
- При $j \neq k$: σ_{jk} — ковариация между признаками j и k (мера линейной связи)

Значения ковариации:

- $\sigma_{jk} > 0$ — признаки положительно коррелированы (растут вместе)
- $\sigma_{jk} < 0$ — признаки отрицательно коррелированы (один растёт, другой убывает)
- $\sigma_{jk} \approx 0$ — признаки некоррелированы (линейно независимы)

Структура ковариационной матрицы:

$$\Sigma = \begin{bmatrix} \sigma_{11} & \sigma_{12} & \cdots & \sigma_{1d} \\ \sigma_{21} & \sigma_{22} & \cdots & \sigma_{2d} \\ \vdots & \vdots & \ddots & \vdots \\ \sigma_{d1} & \sigma_{d2} & \cdots & \sigma_{dd} \end{bmatrix}$$

Диагональ содержит дисперсии, недиагональные элементы — ковариации.

Свойства ковариационной матрицы:

- Симметричная: $\Sigma = \Sigma^T$ (так как $\sigma_{jk} = \sigma_{kj}$)
- Положительно полуопределённая: $v^T \Sigma v \geq 0$ для любого вектора v
- Диагональные элементы — дисперсии: $\sigma_{jj} = \text{Var}(x_j) \geq 0$
- Недиагональные элементы — ковариации: $\sigma_{jk} = \text{Cov}(x_j, x_k)$
- Собственные значения неотрицательны: $\lambda_i \geq 0$ для всех i

Пример расчёта (2D случай):
Пусть центрированные данные:

$$X_{\text{centered}} = \begin{bmatrix} -1 & -1 \\ 0 & 0 \\ 1 & 1 \end{bmatrix}$$

Тогда:

$$\Sigma = \frac{1}{2} \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} \begin{bmatrix} -1 & -1 \\ 0 & 0 \\ 1 & 1 \end{bmatrix} = \frac{1}{2} \begin{bmatrix} 2 & 2 \\ 2 & 2 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$$

Здесь $\sigma_{11} = \sigma_{22} = 1$ (дисперсии) и $\sigma_{12} = \sigma_{21} = 1$ (ковариация).

21.2.5 Собственные значения и собственные векторы

Собственный вектор v и собственное значение λ матрицы Σ удовлетворяют фундаментальному уравнению:

$$\Sigma v = \lambda v$$

Это уравнение означает: умножение матрицы Σ на вектор v эквивалентно умножению этого вектора на скаляр λ.

Геометрическая интерпретация: Собственный вектор — это направление, вдоль которого линейное преобразование (задаваемое матрицей Σ) только растягивает или сжимает вектор, но не поворачивает его.

Как найти собственные значения:
Из уравнения $\Sigma v = \lambda v$ получаем $(\Sigma - \lambda I)v = 0$, где I — единичная матрица.
Нетривиальное решение существует, если:

$$\det(\Sigma - \lambda I) = 0$$

Это характеристическое уравнение — полином степени d относительно λ.

Интерпретация в контексте PCA:

- Собственный вектор v_i — направление i-й главной компоненты в исходном пространстве признаков
- Собственное значение λ_i — дисперсия данных вдоль направления v_i (величина разброса в этом направлении)
- Большое λ_i означает, что в направлении v_i данные сильно варьируются (это важное направление)
- Малое λ_i означает, что в направлении v_i данные почти не меняются (это направление несёт мало информации)

Важное свойство: Собственные векторы симметричной матрицы ортогональны:

$$v_i^T v_j = \begin{cases} 1, & i = j \\ 0, & i \neq j \end{cases}$$

Это означает, что главные компоненты взаимно перпендикулярны и образуют новую ортогональную систему координат.

Пример (для матрицы из предыдущего примера $\Sigma = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$):

Характеристическое уравнение:

$$\det \begin{bmatrix} 1-\lambda & 1 \\ 1 & 1-\lambda \end{bmatrix} = (1-\lambda)^2 - 1 = 0$$

Решая, получаем $\lambda_1 = 2, \lambda_2 = 0$.

Собственные векторы: $v_1 = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ 1 \end{bmatrix}, v_2 = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ -1 \end{bmatrix}$.

21.2.6 Собственное разложение (Eigendecomposition)

Любая симметричная матрица Σ может быть разложена на произведение трёх матриц:

$$\Sigma = V \Lambda V^T$$

где:

- $V = [v_1, v_2, \dots, v_d]$ — ортогональная матрица размера $d \times d$, столбцы которой — собственные векторы
- $\Lambda = \text{diag}(\lambda_1, \lambda_2, \dots, \lambda_d)$ — диагональная матрица размера $d \times d$ с собственными значениями на диагонали

Структура матриц:

$$V = \begin{bmatrix} | & | & & | \\ v_1 & v_2 & \cdots & v_d \\ | & | & & | \end{bmatrix}, \quad \Lambda = \begin{bmatrix} \lambda_1 & 0 & \cdots & 0 \\ 0 & \lambda_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \lambda_d \end{bmatrix}$$

Свойство ортогональности: Матрица V ортогональна, то есть $V^T V = V V^T = I$, где I — единичная матрица. Это означает, что $V^{-1} = V^T$.

Собственные значения упорядочены по убыванию: $\lambda_1 \geq \lambda_2 \geq \dots \geq \lambda_d \geq 0$.

Первое собственное значение λ_1 — наибольшее, соответствующий ему собственный вектор v_1 — направление максимальной дисперсии (первая главная компонента).

Почему eigendecomposition важна для PCA:

Разложение $\Sigma = V \Lambda V^T$ даёт нам:

1. Направления максимальной дисперсии (столбцы V)
2. Величину дисперсии в каждом направлении (элементы Λ)
3. Способ проецировать данные на новые оси ($Y = XV$)

21.3. Алгоритм PCA

21.3.1 Пошаговый алгоритм

Вход: Матрица данных $X \in \mathbb{R}^{n \times d}$, число компонент k
Выход: Проекция $Y \in \mathbb{R}^{n \times k}$, матрица преобразования $W \in \mathbb{R}^{d \times k}$
Шаги:

1. Стандартизировать данные: $X_{\text{std}} \leftarrow \frac{X - \mu}{\sigma}$

   Здесь μ — вектор средних, σ — вектор стандартных отклонений. Вычитание и деление выполняются поэлементно для каждого признака.

2. Вычислить ковариационную матрицу: $\Sigma = \frac{1}{n-1} X_{\text{std}}^T X_{\text{std}}$

   Это матрица размера $d \times d$, которая кодирует все парные взаимосвязи между признаками.

3. Вычислить собственные значения и векторы: $\Sigma = V \Lambda V^T$

   Решить eigenvalue problem. На практике используют библиотечные функции (numpy.linalg.eig или SVD).

4. Сортировать собственные значения по убыванию: $\lambda_1 \geq \lambda_2 \geq \ldots \geq \lambda_d$

   Переупорядочить столбцы V соответственно. Теперь первый столбец v_1 соответствует наибольшей дисперсии.

5. Выбрать k собственных векторов с наибольшими λ_i: $W = [v_1, v_2, \ldots, v_k]$

   Матрица W размера $d \times k$ содержит k главных компонент.

6. Спроецировать данные: $Y = X_{\text{std}} W$

   Умножение матрицы $(n \times d)$ на матрицу $(d \times k)$ даёт матрицу $(n \times k)$ — координаты всех n объектов в пространстве k главных компонент.

7. Вернуть Y, W

21.3.2 Проекция на главные компоненты

Матрица преобразования $W \in \mathbb{R}^{d \times k}$ состоит из k собственных векторов:

$$W = \begin{bmatrix} | & | & & | \\ v_1 & v_2 & \cdots & v_k \\ | & | & & | \end{bmatrix}$$

Каждый столбец v_i — это вектор весов, определяющий, как исходные признаки комбинируются для получения i-й главной компоненты.

Проекция данных на новое пространство:

$$Y = X_{\text{std}} W$$

В развёрнутом виде для одного объекта $x \in \mathbb{R}^d$:

$$y = W^T x = \begin{bmatrix} v_1^T x \\ v_2^T x \\ \vdots \\ v_k^T x \end{bmatrix} \in \mathbb{R}^k$$

Каждая компонента $y_i = v_i^T x$ — это скалярное произведение исходного вектора признаков на направление i-й главной компоненты.

Каждый столбец Y — значения одной главной компоненты для всех объектов.

21.3.3 Восстановление данных

Из пространства главных компонент можно вернуться к исходному пространству:

$$X_{\text{reconstructed}} = YW^T$$

Это обратное преобразование. Если $k = d$ (использовали все компоненты), то $X_{\text{reconstructed}} = X_{\text{std}}$ (точное восстановление).

Если $k < d$, то происходит потеря информации.

Ошибка реконструкции:

$$\text{MSE} = \frac{1}{nd}\sum_{i=1}^{n}\sum_{j=1}^{d}(x_{ij}^{\text{std}} - x_{ij}^{\text{reconstructed}})^2$$

Чем больше k (число компонент), тем меньше ошибка. При $k = d$ ошибка равна 0.

Связь с отброшенными компонентами:

Ошибка реконструкции равна сумме отброшенных собственных значений:

$$\text{MSE} = \frac{1}{n}\sum_{i=k+1}^{d}\lambda_i$$

21.4. Выбор числа компонент

21.4.1 Объясняемая дисперсия

Доля дисперсии, объясняемая i-й компонентой:

$$\text{Explained Variance Ratio}_i = \frac{\lambda_i}{\sum_{j=1}^{d}\lambda_j}$$

Это показывает, какую долю от общей вариабельности данных несёт i-я компонента.

Кумулятивная объясняемая дисперсия первых k компонент:

$$\text{Cumulative Explained Variance}_k = \frac{\sum_{i=1}^{k}\lambda_i}{\sum_{j=1}^{d}\lambda_j}$$

Это доля информации, сохраняемая при использовании k компонент.

21.4.2 Правило порога дисперсии

Выбрать k такое, что кумулятивная объясняемая дисперсия $\geq$ порог (обычно 80%, 90%, 95%):

$$k = \min\left\{m : \frac{\sum_{i=1}^{m}\lambda_i}{\sum_{j=1}^{d}\lambda_j} \geq \text{threshold}\right\}$$

Типичные пороги:

- 80% — быстрое приближение
- 90% — баланс между сжатием и точностью
- 95% — высокая точность
- 99% — почти полное сохранение информации

21.4.3 Scree Plot (график осыпи)

График собственных значений λ_i по убыванию помогает визуально определить оптимальное число компонент.

Правило локтя: Выбрать k в точке, где график переходит в пологую часть (резкий переход к плавному спаду). Эта точка называется "локтем" (elbow).

Интерпретация: После точки локтя добавление новых компонент даёт малый прирост объясняемой дисперсии.

21.4.4 Кросс-валидация

Для задач с учителем — выбрать k, максимизирующее качество модели на валидационной выборке:

1. Для $k = 1, 2, \ldots, d$:
   - Применить PCA с k компонентами на обучающей выборке
   - Обучить модель на редуцированных данных
   - Оценить качество на валидации
2. Выбрать k с максимальным качеством

21.5. Связь PCA и SVD

21.5.1 Сингулярное разложение (SVD)

Сингулярное разложение (Singular Value Decomposition, SVD) — это фундаментальная факторизация матрицы, которая обобщает eigendecomposition на прямоугольные матрицы.

Любую матрицу $X \in \mathbb{R}^{n \times d}$ (не обязательно квадратную!) можно разложить как:

$$X = U \Sigma V^T$$

где:

- $U \in \mathbb{R}^{n \times n}$ — ортогональная матрица левых сингулярных векторов ($U^T U = I$)
- $\Sigma \in \mathbb{R}^{n \times d}$ — диагональная (прямоугольная) матрица сингулярных значений $\sigma_i \geq 0$
- $V \in \mathbb{R}^{d \times d}$ — ортогональная матрица правых сингулярных векторов ($V^T V = I$)

Структура разложения:

$$\underset{n \times d}{X} = \underset{n \times n}{U} \; \underset{n \times d}{\Sigma} \; \underset{d \times d}{V^T}$$

Матрица Σ имеет вид (для случая $n > d$):

$$\Sigma = \begin{bmatrix} \sigma_1 & 0 & \cdots & 0 \\ 0 & \sigma_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \sigma_d \\ 0 & 0 & \cdots & 0 \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \cdots & 0 \end{bmatrix}$$

где $\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_d \geq 0$ — сингулярные значения, упорядоченные по убыванию.

Интерпретация компонент SVD:

- Столбцы U — левые сингулярные векторы, описывают пространство объектов
- Столбцы V — правые сингулярные векторы, описывают пространство признаков
- Сингулярные значения σ_i — "сила" каждой компоненты разложения

Reduced SVD:

На практике часто используют усечённое (reduced) SVD, где оставляют только $r = \min(n, d)$ ненулевых сингулярных значений:

$$X = U_r \Sigma_r V_r^T$$

где $U_r \in \mathbb{R}^{n \times r}$, $\Sigma_r \in \mathbb{R}^{r \times r}$, $V_r \in \mathbb{R}^{d \times r}$.

21.5.2 Связь SVD и PCA

SVD и PCA тесно связаны. Для центрированных данных X_{centered} применение SVD даёт PCA напрямую, без вычисления ковариационной матрицы.

Теорема: Пусть $X_{\text{centered}} = U\Sigma V^T$ — SVD разложение центрированной матрицы данных. Тогда:

1. Столбцы V — собственные векторы ковариационной матрицы $\Sigma_{\text{cov}} = \frac{1}{n-1} X_{\text{centered}}^T X_{\text{centered}}$ (главные компоненты)
2. Сингулярные значения σ_i связаны с собственными значениями:

$$\lambda_i = \frac{\sigma_i^2}{n-1}$$

где λ_i — собственные значения ковариационной матрицы.

3. Проекция данных на главные компоненты: $Y = U\Sigma$ (первые k столбцов)

Альтернативно: $Y = X_{\text{centered}} V$ (что эквивалентно $U\Sigma$)

Доказательство связи:

Ковариационная матрица:

$$\Sigma_{\text{cov}} = \frac{1}{n-1} X_{\text{centered}}^T X_{\text{centered}}$$

Подставим $X_{\text{centered}} = U\Sigma V^T$:

$$\Sigma_{\text{cov}} = \frac{1}{n-1} (U\Sigma V^T)^T (U\Sigma V^T) = \frac{1}{n-1} V\Sigma^T U^T U\Sigma V^T$$

Так как $U^T U = I$:

$$\Sigma_{\text{cov}} = \frac{1}{n-1} V\Sigma^T \Sigma V^T = V \left(\frac{\Sigma^T \Sigma}{n-1} \right) V^T$$

Матрица $\Sigma^T \Sigma$ диагональна с элементами σ_i^2 на диагонали, поэтому:

$$\Sigma_{\text{cov}} = V \Lambda V^T$$

где $\Lambda = \text{diag}\left(\frac{\sigma_1^2}{n-1}, \frac{\sigma_2^2}{n-1}, \ldots, \frac{\sigma_d^2}{n-1} \right)$.

Это eigendecomposition ковариационной матрицы, где столбцы V — собственные векторы, а $\lambda_i = \frac{\sigma_i^2}{n-1}$ — собственные значения.

Преимущество SVD:

- Численно более стабилен и быстрее для больших матриц
- Не требует явного вычисления ковариационной матрицы (экономия памяти)
- Работает для матриц любого размера ($n \gg d$ или $n \ll d$)

Важно: Большинство библиотек (sklearn, numpy) используют SVD, а не eigendecomposition!
Вычислительная сложность:

- Eigendecomposition ковариационной матрицы: $O(d^3)$ (после вычисления Σ_{cov} за $O(nd^2)$)
- SVD напрямую: $O(\min(n^2 d, nd^2))$
- Для $n \gg d$: SVD выгоднее (вычисляется на исходной матрице, не на $d \times d$ ковариационной)

21.6. Визуализация с PCA

21.6.1 Проекция на 2D

Для визуализации многомерных данных используют проекцию на первые 2 компоненты:

- $k = 2$
- Получить $Y \in \mathbb{R}^{n \times 2}$
- Построить scatter plot: PC1 vs PC2, цвет по классам

Интерпретация: Точки одного класса должны образовывать кластеры на плоскости главных компонент.

21.6.2 Biplot

Biplot — график, совмещающий:

- Проекцию объектов на PC1 и PC2 (точки)
- Вклад исходных признаков (стрелки)

Веса (loadings):

$$\text{loading}_{ij} = v_{ij}\sqrt{\lambda_i}$$

где v_{ij} — элемент i-го собственного вектора.
Интерпретация стрелок:

- Длина стрелки $\propto$ влиянию признака на компоненту
- Направление стрелки — направление наибольшего увеличения признака
- Угол между стрелками — корреляция между признаками

21.6.3 Интерпретация компонент

Веса признаков в главной компоненте:

$$\text{PC}_i = w_{i1}x_1 + w_{i2}x_2 + \ldots + w_{id}x_d$$

где w_{ij} — элементы собственного вектора v_i.
Признаки с большими $|w_{ij}|$ сильно влияют на PC_i.

21.7. Продвинутые методы

21.7.1 Kernel PCA

Для нелинейно разделимых данных используют **Kernel PCA**:
Идея:

- Применяют kernel trick (как в SVM)
- Неявно переводят данные в высокоразмерное пространство через отображение $\varphi : \mathbb{R}^d \to \mathbb{R}^D$
- Затем применяют PCA в этом пространстве

Ядровая функция:

$$K(x, y) = \varphi(x)^T \varphi(y)$$

Популярные ядра:

1. RBF (Гауссово):

$$K(x, y) = \exp\left(-\gamma \|x - y\|^2\right)$$

2. Полиномиальное:

$$K(x, y) = (x^T y + c)^d$$

3. Сигмоидальное:

$$K(x, y) = \tanh(\alpha x^T y + c)$$

Когда использовать: Данные образуют нелинейные структуры (полумесяцы, окружности).

21.7.2 Incremental PCA

Для очень больших данных, не влезающих в память:
Идея:

- Разбить данные на батчи (mini-batches)
- Обновлять главные компоненты инкрементально
- Не требует загрузки всех данных в память

Алгоритм:

1. Инициализировать пустую модель PCA
2. Для каждого батча:
   - Обновить статистики (средние, ковариации)
   - Обновить собственные векторы
3. Получить финальные главные компоненты

Применение: Потоковые данные, онлайн-обучение, датасеты > 10 GB.

21.7.3 Sparse PCA

Добавляет L1-регуляризацию для получения разреженных весов:

$$\min_W \|X - XWW^T\|_F^2 + \alpha\|W\|_1$$

Результат:

- Многие веса $w_{ij} = 0$
- Легче интерпретировать компоненты
- Каждая компонента зависит от малого числа признаков

Применение: Генетика, текстовый анализ, высокоразмерные данные с тысячами признаков.

21.8. Применение PCA

21.8.1 Предобработка для классификации

Типичный pipeline:

1. Стандартизация данных
2. PCA для понижения размерности
3. Классификатор (Logistic Regression, SVM, Random Forest)

Преимущества:

- Быстрее обучение (меньше признаков)
- Меньше переобучения при $k \ll d$
- Устранение мультиколлинеарности
- Снижение шума в данных

Недостатки:

- Может отбросить информацию, важную для классификации
- Компоненты не учитывают метки классов (unsupervised)
- Потеря интерпретируемости признаков

21.8.2 Шумоподавление

PCA отбрасывает компоненты с малой дисперсией $\Rightarrow$ удаление шума:
Алгоритм:

1. Применить PCA с $k < d$ компонентами
2. Восстановить данные: $X_{\text{denoised}} = YW^T$

Применение: Обработка изображений, фильтрация сигналов, очистка измерений.

21.8.3 Компрессия данных

Хранить $Y \in \mathbb{R}^{n \times k}$ и $W \in \mathbb{R}^{d \times k}$ вместо $X \in \mathbb{R}^{n \times d}$:
Степень сжатия:

$$\text{Compression Ratio} = \frac{nk + dk}{nd} = \frac{k(n+d)}{nd}$$

При $k \ll d$ и $n \gg d$: Compression Ratio $\approx \frac{k}{d}$
Пример: Изображения $28 \times 28 = 784$ признака $\Rightarrow$ 50 компонент дают сжатие в 15.7 раз.

21.8.4 Обнаружение выбросов

Объекты с большой ошибкой реконструкции — потенциальные выбросы (аномалии):

$$\text{Anomaly Score}_i = \|x_i - x_i^{\text{reconstructed}}\|^2$$

Критерий: Если Anomaly Score$_i$ > threshold $\Rightarrow$ объект i — выброс.
Пороговое значение: Обычно используют перцентиль, например, 95-й или 99-й.

21.9. Ограничения PCA

21.9.1 Линейность

PCA — линейный метод, не может обнаружить нелинейные зависимости:
Проблема: Если данные лежат на нелинейном многообразии (например, сфера, тор), PCA даст плохое приближение.
Решение:

- Kernel PCA — нелинейное обобщение
- Автоэнкодеры — нейросетевой подход
- t-SNE, UMAP — для визуализации

21.9.2 Чувствительность к масштабу

PCA чувствителен к масштабу признаков:
Проблема: Признаки с большим разбросом значений доминируют в главных компонентах.
Решение: **Обязательно** стандартизировать данные перед PCA.

21.9.3 Интерпретируемость

Главные компоненты — линейные комбинации исходных признаков:

$$\text{PC}_1 = 0.45 \cdot \text{height} + 0.32 \cdot \text{weight} + 0.67 \cdot \text{age} + \ldots$$

Проблема: Сложно интерпретировать смысл компонент.
Решение:

- Sparse PCA — разреженные веса
- Анализ весов (loadings) — выявление доминирующих признаков
- Biplot — визуализация вклада признаков

21.9.4 Unsupervised метод

PCA не использует метки классов:

Проблема: Может отбросить признаки, важные для разделения классов, но имеющие малую дисперсию.

Пример: В задаче классификации рака может быть один ген с малой вариацией, но критически важный для диагностики.

Альтернатива: LDA (Linear Discriminant Analysis) — supervised метод, максимизирующий разделимость классов.

21.10. Практическая реализация с Scikit-learn

21.10.1 Базовое применение

```
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler

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

# PCA с 2 компонентами
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X_scaled)

# Результаты
print("Объясняемая дисперсия:", pca.explained_variance_ratio_)
print("Главные компоненты:", pca.components_)
```

21.10.2 Выбор числа компонент по порогу дисперсии

```
# Оставить 95% дисперсии
pca = PCA(n_components=0.95)
X_pca = pca.fit_transform(X_scaled)

print(f"Исходная размерность: {X_scaled.shape[1]}")
print(f"Новая размерность: {X_pca.shape[1]}")
print(f"Объясняемая дисперсия:
 ↪  {pca.explained_variance_ratio_.sum():.4f}")
```

21.10.3 Визуализация Scree Plot

```
import matplotlib.pyplot as plt

# PCA со всеми компонентами
pca_full = PCA()
pca_full.fit(X_scaled)

# Кумулятивная дисперсия
explained_var = pca_full.explained_variance_ratio_
cumsum_var = np.cumsum(explained_var)

```

```
# График
plt.figure(figsize=(10, 5))
plt.plot(range(1, len(cumsum_var)+1), cumsum_var, 'o-')
plt.axhline(y=0.95, color='r', linestyle='--', label='95%')
plt.xlabel('Число компонент')
plt.ylabel('Кумулятивная объясняемая дисперсия')
plt.legend()
plt.show()
```

21.10.4 Pipeline с классификацией

```
from sklearn.pipeline import Pipeline
from sklearn.linear_model import LogisticRegression

# Pipeline: Scaler → PCA → Classifier
pipeline = Pipeline([
    ('scaler', StandardScaler()),
    ('pca', PCA(n_components=0.95)),
    ('classifier', LogisticRegression())
])

# Обучение на исходных данных
pipeline.fit(X_train, y_train)

# Предсказание
y_pred = pipeline.predict(X_test)
print(f"Accuracy: {accuracy_score(y_test, y_pred):.4f}")
```

21.10.5 Kernel PCA

```
from sklearn.decomposition import KernelPCA

# Kernel PCA с RBF ядром
kpca = KernelPCA(n_components=2, kernel='rbf', gamma=0.1)
X_kpca = kpca.fit_transform(X_scaled)

# Другие ядра: 'linear', 'poly', 'sigmoid', 'cosine'
```

21.10.6 Incremental PCA

```
from sklearn.decomposition import IncrementalPCA

# Incremental PCA для больших данных
ipca = IncrementalPCA(n_components=20, batch_size=500)

# Обучение батчами
n_batches = X_large.shape[0] // 500
for batch_idx in range(n_batches):
    start = batch_idx * 500
    end = start + 500
    ipca.partial_fit(X_large[start:end])
```

```
12
13 # Трансформация
14 X_ipca = ipca.transform(X_large)
```

21.10.7 Восстановление данных

```
1  # Прямое преобразование
2  pca = PCA(n_components=50)
3  X_pca = pca.fit_transform(X_scaled)
4
5  # Обратное преобразование (восстановление)
6  X_reconstructed = pca.inverse_transform(X_pca)
7
8  # Ошибка реконструкции
9  mse = np.mean((X_scaled - X_reconstructed) ** 2)
10 print(f"Ошибка реконструкции (MSE): {mse:.6f}")
```

21.11. Примеры применения

21.11.1 Пример 1: Датасет Iris (4D 2D)

Задача: Визуализация 4D данных Iris на 2D плоскости.
Данные: 150 образцов, 4 признака (длина и ширина чашелистика и лепестка), 3 класса цветков.
Результат: Первые 2 компоненты объясняют 95.8% дисперсии, классы хорошо разделены на плоскости PC1-PC2.

21.11.2 Пример 2: MNIST Digits (784D 50D)

Задача: Ускорение классификации рукописных цифр.
Данные: Изображения $28 \times 28 = 784$ пикселя.
Результаты:

- Без PCA: Accuracy = 0.91, время обучения 10 сек
- С PCA (50 компонент): Accuracy = 0.89, время обучения 2 сек

Вывод: Небольшая потеря точности (2%), значительное ускорение (5x).

21.11.3 Пример 3: Eigenfaces (распознавание лиц)

Задача: Распознавание лиц на изображениях $64 \times 64 = 4096$ пикселей.
Метод:

- Применить PCA к набору изображений лиц
- Первые 150 главных компонент — "eigenfaces" (собственные лица)
- Каждое лицо представляется как линейная комбинация eigenfaces

Результат: Сжатие в 27 раз, сохранение 95% информации, точность распознавания > 90%.

21.12. Выводы и рекомендации

21.12.1 Когда использовать PCA

- Высокоразмерные данные ($d > 50$ признаков)
- Визуализация многомерных данных
- Предобработка перед классификацией или регрессией
- Устранение мультиколлинеарности между признаками
- Шумоподавление в данных
- Компрессия данных для хранения или передачи
- Ускорение обучения моделей машинного обучения

21.12.2 Когда НЕ использовать PCA

- Признаки уже некоррелированы и независимы
- Нелинейные зависимости в данных (использовать Kernel PCA)
- При работе с изображениями (теряется пространственная связь)
- Интерпретируемость модели критична
- Малое число признаков ($d < 10$)
- Категориальные признаки (требуется предобработка)

21.12.3 Лучшие практики

1. Всегда стандартизировать данные перед PCA
2. Использовать Scree Plot для выбора числа компонент
3. Проверить качество модели с/без PCA на валидации
4. Визуализировать первые 2-3 компоненты
5. Для нелинейных данных использовать Kernel PCA
6. Для очень больших данных использовать Incremental PCA
7. Документировать долю объясняемой дисперсии
8. Сохранять объект PCA для применения к новым данным

22. Основы глубокого обучения. Нейронные сети

22.1. Введение: нейросети как обучаемые функции

Нейронные сети — один из самых мощных инструментов современного машинного обучения. Если в классических моделях (логистическая регрессия, деревья решений, бустинг, SVM) мы заранее задаём форму зависимости, то в нейросетях мы строим *гибкое семейство функций* и учим его по данным.

С математической точки зрения нейросеть — это параметрическая функция

$$f_\theta : \mathbb{R}^d \to \mathbb{R}^m,$$

где θ — все обучаемые параметры (веса и смещения во всех слоях).

Задача обучения обычно формулируется как минимизация среднего значения функции потерь на обучающей выборке:

$$\min_\theta \ \frac{1}{N}\sum_{i=1}^{N} \ell(f_\theta(x_i),\, y_i).$$

Здесь (x_i, y_i) — примеры из данных, ℓ — функция потерь (насколько плохо предсказание), N — число объектов.

22.2. Искусственный нейрон

Определение 22.1. Искусственный нейрон

Искусственный нейрон — вычислительная единица, которая сначала вычисляет линейную комбинацию входов (взвешенную сумму + смещение), а затем применяет нелинейную функцию активации:

$$z = w^T x + b, \qquad a = \sigma(z).$$

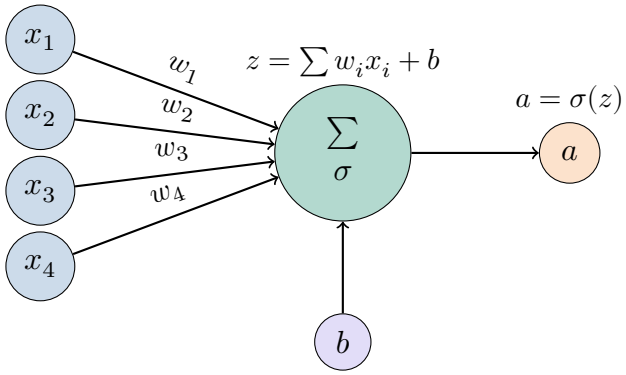


Рис. 25: Искусственный нейрон как «линейное преобразование + нелинейность»

Искусственный нейрон принимает вход $x = (x_1, \dots, x_d)$, вычисляет линейную комбинацию

$$z = \sum_{i=1}^{d} w_i x_i + b = w^T x + b,$$

и применяет функцию активации σ (например, ReLU), чтобы получить выход a.

Замечание 6. Почему нужна нелинейность?

Если бы σ была тождественной ($\sigma(z) = z$), то любая композиция таких «нейронов» сводилась бы к одной линейной функции. Нелинейность позволяет моделировать сложные зависимости.

22.3. От нейрона к слою: векторизация и размеры

Один нейрон — это скалярный выход. На практике мы почти всегда используем **слой** из многих нейронов.

Пусть вход $x \in \mathbb{R}^d$, а в слое m нейронов. Тогда:

$$z = Wx + b, \qquad a = \sigma(z),$$

где

$$W \in \mathbb{R}^{m \times d}, \quad b \in \mathbb{R}^m, \quad z, a \in \mathbb{R}^m.$$

Функция σ применяется *покомпонентно*:

$$a_j = \sigma(z_j), \qquad j = 1, \dots, m.$$

Замечание 7. Интуиция

Матрица W — это «набор» из m весовых векторов (по одному на нейрон). Умножение Wx одновременно считает взвешенные суммы для всех нейронов слоя.

22.4. Архитектура полносвязной нейронной сети (MLP)

Определение 22.2. Полносвязная нейронная сеть (MLP)

Полносвязная нейронная сеть (Multi-Layer Perceptron, MLP) — это композиция слоёв вида $a = \phi(Wx + b)$, где ϕ — нелинейность (или другая функция на выходе).

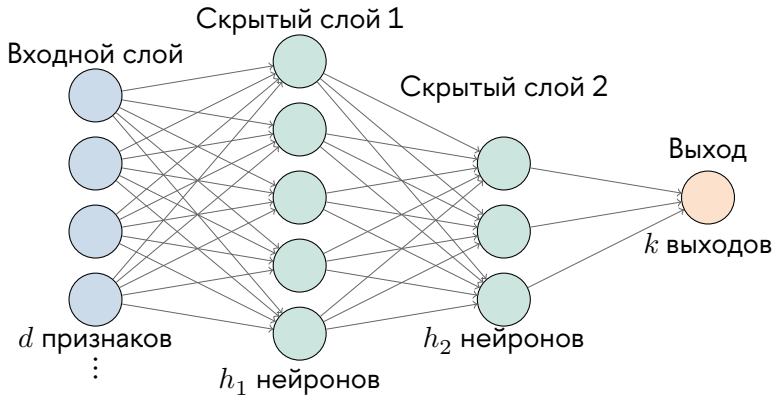


Рис. 26: Архитектура MLP: входной слой, несколько скрытых слоёв, выход

Стандартная запись для сети из L слоёв:

$$a^{(0)} = x,$$

$$z^{(l)} = W^{(l)} a^{(l-1)} + b^{(l)}, \qquad a^{(l)} = \phi^{(l)}(z^{(l)}), \qquad l = 1, \dots, L.$$

На последнем слое $\phi^{(L)}$ выбирается под задачу (регрессия, классификация, кластеризация).

22.5. Функции активации: формулы, производные, свойства

Функции активации добавляют нелинейность, но также сильно влияют на градиенты и скорость обучения.

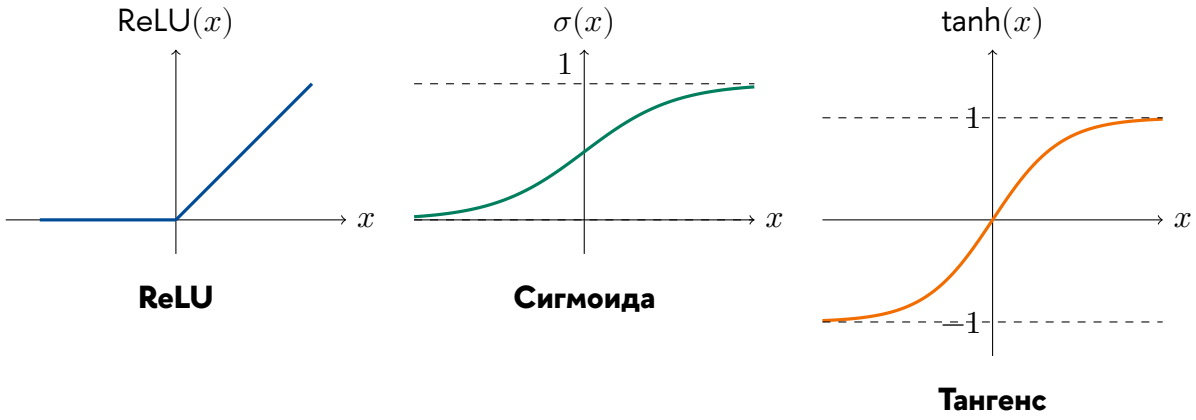


Рис. 27: Популярные функции активации

ReLU (Rectified Linear Unit):

$$\mathrm{ReLU}(x) = \max(0, x).$$

Производная (в точке 0 можно считать 0 или 1 — на практике неважно):

$$\mathrm{ReLU}'(x) = \begin{cases} 1, & x > 0, \\ 0, & x < 0. \end{cases}$$

Плюс: для $x > 0$ производная равна 1, поэтому градиенты не «умирают» из-за самой активации. **Минус:** если нейрон почти всегда видит отрицательные z, то его градиент часто 0 и он может перестать обучаться.

Leaky ReLU:

$$\mathrm{LeakyReLU}(x) = \max(\alpha x, x), \qquad \alpha \approx 0.01.$$

Даёт небольшой градиент и при $x < 0$, снижая риск «умирающих» нейронов.

Сигмоида (sigmoid):

$$\sigma(x) = \frac{1}{1 + e^{-x}}, \qquad \sigma'(x) = \sigma(x)(1 - \sigma(x)).$$

Важно: при больших $|x|$ сигмоида насыщается (стремится к 0 или 1), а производная становится близка к нулю — это ведёт к затухающим градиентам.

Гиперболический тангенс (tanh):

$$\tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}, \qquad \tanh'(x) = 1 - \tanh^{2}(x).$$

Тоже насыщается при больших $|x|$, но центрирован около нуля, что иногда ускоряет обучение относительно сигмоиды.

Замечание 8. Выбор активации

В современных MLP для скрытых слоёв чаще используют ReLU и её вариации (Leaky ReLU, ELU, GELU). Сигмоиду обычно оставляют для выхода в бинарной классификации.

22.6. Обратное распространение ошибки (backpropagation): строгая и удобная матричная форма

Обратное распространение ошибки (backpropagation) — это алгоритм, который позволяет эффективно вычислять градиенты функции потерь по всем параметрам нейросети, используя цепное правило.

Идея проста: сначала мы делаем *прямой проход* (forward pass) и считаем все промежуточные значения, а затем делаем *обратный проход* (backward pass) и считаем градиенты, двигаясь от функции потерь к входу сети.

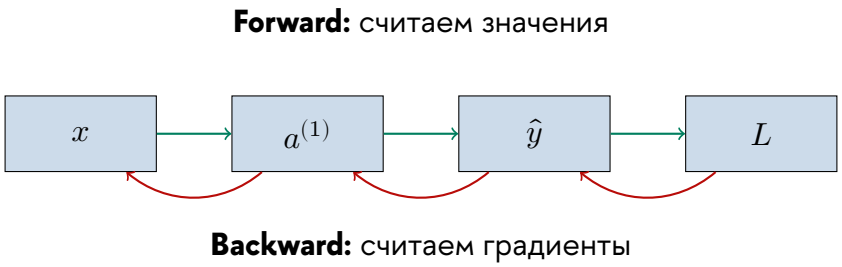


Рис. 28: Прямой и обратный проходы в нейросети

Теорема 22.1. Цепное правило

Если $z = f(y)$ и $y = g(x)$, то:

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y} \cdot \frac{\partial y}{\partial x}.$$

Для композиции многих функций градиент получается произведением (или последовательным применением) локальных производных.

22.6.1 Один слой: универсальные формулы

Рассмотрим один слой (для одного объекта):

$$z = Wa + b, \qquad h = \phi(z),$$

и пусть мы уже знаем градиент $\delta_h = \frac{\partial L}{\partial h}$ (он приходит «сверху», от следующих слоёв).

Тогда:

$$\delta_z = \frac{\partial L}{\partial z} = \delta_h \odot \phi'(z),$$

где $\odot$ — поэлементное умножение.

А градиенты по параметрам слоя и по входу:

$$\frac{\partial L}{\partial W} = \delta_z\, a^T, \qquad \frac{\partial L}{\partial b} = \delta_z, \qquad \frac{\partial L}{\partial a} = W^T \delta_z.$$

Замечание 9. Почему это важно?

Эти 4 формулы — «кирпичики» обратного распространения ошибки для любой глубины. Вычислительная сложность линейна по числу слоёв: мы один раз идём вперёд, один раз назад.

22.7. Логиты: что это такое и почему они нужны

Определение 22.3. Логит

Логит — это «сырой» выход модели *до* преобразования в вероятность.

В задачах классификации модель часто сначала выдаёт число (или вектор чисел), которое удобно интерпретировать как *оценку уверенности*, а затем превращать в вероятность.

Бинарная классификация. Пусть модель выдаёт число $z \in \mathbb{R}$. Тогда вероятность класса 1 получается применением сигмоиды:

$$\hat{y} = \sigma(z).$$

Число z и есть *логит*. Если z большой и положительный, то $\hat{y}$ близко к 1; если большой и отрицательный — $\hat{y}$ близко к 0.

Многоклассовая классификация. Пусть модель выдаёт вектор $z \in \mathbb{R}^k$. Тогда вероятности по классам получаются применением softmax:

$$\hat{p} = \mathrm{softmax}(z).$$

Компоненты $z_1, \dots, z_k$ называются *логитами* классов.

Замечание 10. Почему loss в PyTorch обычно принимает логиты

Многие функции потерь реализованы так, чтобы принимать логиты и внутри делать численно устойчивые преобразования (например, `BCEWithLogitsLoss` и `CrossEntropyLoss`). Это уменьшает риск переполнений и улучшает стабильность обучения.

22.7.1 Пример: бинарная классификация (логиты, сигмоида и BCE)

Пусть последний слой даёт число $z \in \mathbb{R}$ (логит), а вероятность класса 1:

$$\hat{y} = \sigma(z).$$

Пусть истинная метка $y \in \{0, 1\}$. Функция потерь (BCE):

$$L = -(y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})).$$

Тогда (ключевой удобный факт):

$$\frac{\partial L}{\partial z} = \hat{y} - y.$$

Это означает, что «ошибка» на логите — просто разница между предсказанной вероятностью и истинной меткой.

Замечание 11. Практический вывод

В реализациях обычно не считают σ отдельно: используют численно устойчивую функцию `BCEWithLogitsLoss`, которая внутри объединяет сигмоиду и BCE.

22.7.2 softmax и кросс-энтропия: градиент $\hat{p} - y$

Для многоклассовой классификации пусть сеть выдаёт логиты $z \in \mathbb{R}^k$, а вероятности:

$$\hat{p} = \mathrm{softmax}(z), \qquad \hat{p}_i = \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_j}}.$$

Пусть y — one-hot вектор истинного класса (то есть $y_c = 1$, остальные 0). Тогда кросс-энтропия:

$$L = -\sum_{i=1}^{k} y_i \log(\hat{p}_i).$$

Ключевой результат:

$$\frac{\partial L}{\partial z} = \hat{p} - y.$$

Он объясняет, почему `CrossEntropyLoss` в PyTorch принимает *логиты*, а не вероятности: внутри всё делается стабильно и градиент получается простым.

22.8. Проблемы градиентов: затухание и взрыв

При обучении глубоких сетей градиенты вычисляются по цепному правилу и часто содержат произведение многих матриц (и производных активаций). Поэтому возможно два сценария:

Затухающий градиент (vanishing): нормы градиентов становятся очень малыми при переходе от выхода к ранним слоям. Тогда первые слои почти не обучаются.

Взрывающийся градиент (exploding): нормы градиентов резко растут, обучение становится нестабильным, появляются `NaN`.

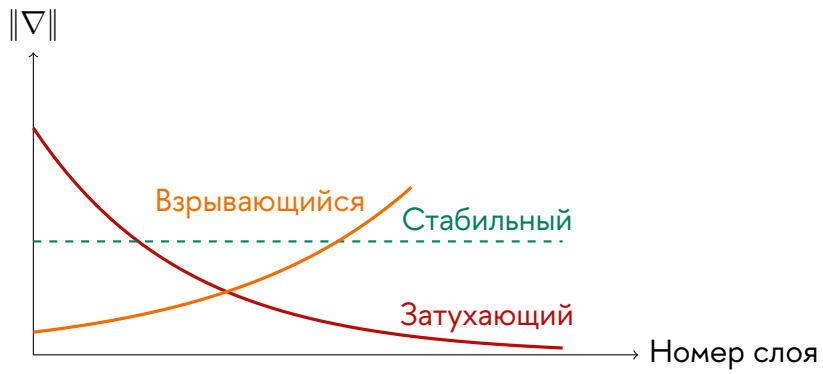


Рис. 29: Типичное поведение норм градиента по глубине сети

Что обычно помогает:

- выбор активации (ReLU/Leaky ReLU вместо сигмоиды/тангенса в скрытых слоях),
- правильная инициализация весов (Xavier/He),
- нормализации (BatchNorm),
- gradient clipping,
- архитектурные приёмы (например, residual connections в более сложных сетях).

22.9. Инициализация весов: почему Xavier и He

Плохая инициализация может быстро привести к затухающим/взрывающимся сигналам. Идея Xavier/He: подобрать дисперсию весов так, чтобы дисперсия активаций (и/или градиентов) не «уплывала» по глубине.

Xavier (Glorot) Initialization:

$$W \sim \mathcal{U}\left(-\sqrt{\frac{6}{n_{\text{in}}+n_{\text{out}}}}, \sqrt{\frac{6}{n_{\text{in}}+n_{\text{out}}}}\right).$$

Часто хорошо для тангенса/сигмоиды.

He Initialization (для ReLU-подобных):

$$W \sim \mathcal{N}\left(0, \frac{2}{n_{\text{in}}}\right).$$

Замечание 12. Практика

В PyTorch многие слои уже инициализируются разумно, но полезно знать, как вручную применить `nn.init.kaiming_normal_` или `nn.init.xavier_uniform_`.

22.10. Batch Normalization

BatchNorm нормализует активации внутри мини-батча:

$$\hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}, \qquad y_i = \gamma \hat{x}_i + \beta.$$

Это стабилизирует обучение и позволяет использовать большие learning rate.

Важно: в режиме `train` используются статистики батча, в режиме `eval` — накопленные running-статистики.

22.11. Оптимизация: SGD, Momentum, Adam/AdamW

После вычисления градиентов обратным распространением ошибки нужно обновить параметры.

SGD (градиентный спуск):

$$\theta \leftarrow \theta - \eta \nabla_\theta L,$$

где η — learning rate.

Momentum: накапливает «скорость» и сглаживает траекторию:

$$v_t = \beta v_{t-1} + (1-\beta)\nabla_\theta L, \qquad \theta_t = \theta_{t-1} - \eta v_t.$$

Adam: хранит экспоненциальные средние градиента и квадрата градиента:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t, \qquad v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2,$$

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}, \qquad \hat{v}_t = \frac{v_t}{1-\beta_2^t},$$

$$\theta_t = \theta_{t-1} - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}.$$

Типичные параметры: $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$.

AdamW: корректно реализует weight decay как отдельный шаг, что часто работает лучше, чем «L2 внутри Adam».

22.12. Gradient Clipping

Ограничивает норму градиента, чтобы избежать взрыва:

$$g := \begin{cases} g, & \|g\| \le T, \\ \frac{T}{\|g\|} g, & \|g\| > T. \end{cases}$$

22.13. Введение в PyTorch

PyTorch — библиотека для глубокого обучения с автоматическим дифференцированием. Главная идея: вы пишете вычисления почти как в NumPy, а PyTorch автоматически строит вычислительный граф и считает градиенты.

Важный тезис: PyTorch — это не только про «нейросети», это про *тензоры + автоматические производные*.

Основные компоненты:

- `torch.Tensor` — многомерные массивы (как NumPy, но могут хранить градиенты и жить на GPU),
- `torch.autograd` — автоматическое дифференцирование,
- `torch.nn` — слои и модели,
- `torch.optim` — оптимизаторы,
- `torch.utils.data` — датасеты и загрузчики батчей.

22.13.1 Тензоры: shape, dtype, device

Тензор — это массив с:

- **shape** (размерности),
- **dtype** (тип данных, обычно `float32`),
- **device** (CPU или GPU).

Примеры:

```
import torch

x = torch.tensor([1.0, 2.0, 3.0])    # float tensor на CPU
A = torch.zeros((3, 4))              # 3x4
W = torch.randn((5, 3))              # случайная матрица
b = torch.zeros((5,))                # bias-вектор

z = W @ x + b                        # матричное умножение +
 ↪ broadcasting
print(z.shape)                       # torch.Size([5])
```

Device (CPU/GPU):

```
device = torch.device("cuda" if torch.cuda.is_available() else
 ↪ "cpu")

W = W.to(device)
x = x.to(device)
b = b.to(device)
z = W @ x + b
```

Замечание 13. Типичная ошибка

Если часть тензоров на CPU, а часть на GPU, PyTorch выдаст ошибку. Держите модель и данные на одном `device`.

22.13.2 Broadcasting и размерности: аккуратность важнее всего

В PyTorch (как и в NumPy) многие операции автоматически «растягивают» размерности. Например, если W имеет размер $[m, d]$, а b имеет размер $[m]$, то при сложении $Wx+b$ вектор b автоматически прибавится к каждому объекту в батче.

Пример с батчем:

```
X = torch.randn(32, 3)        # batch=32, d=3
W = torch.randn(5, 3)         # m=5
b = torch.zeros(5)            # m=5

Z = X @ W.t() + b             # Z: [32, 5]
print(Z.shape)
```

Замечание 14. Правило

Если вы забыли, как устроены размерности в линейном слое: в PyTorch вход в `nn.Linear` имеет форму $[batch, d\_in]$, а выход — $[batch, d\_out]$.

22.13.3 Autograd: как PyTorch считает производные

Если тензор создан с `requires_grad=True`, PyTorch будет отслеживать операции и сможет вычислить градиент.

```
import torch

x = torch.tensor(2.0, requires_grad=True)
y = x**2 + 3*x + 1
y.backward()
print(x.grad)   # dy/dx = 2x + 3 = 7
```

Градиенты накапливаются:

```
x = torch.tensor(2.0, requires_grad=True)
y1 = x**2
y1.backward()
print(x.grad)   # 4

y2 = 3*x
y2.backward()
print(x.grad)   # 4 + 3 = 7 (накопилось)
```

Поэтому в обучении модели делают `optimizer.zero_grad()` на каждом шаге.

`detach` и `no_grad`:

```
with torch.no_grad():
    # граф не строится, память экономится, градиенты не считаются
    y_pred = model(x)

x_det = x.detach()  # новый тензор без связи с графом
```

22.13.4 `nn.Module`, `nn.Sequential` и готовые слои

В PyTorch нейросети удобно собирать из готовых блоков:

- `nn.Linear` — полносвязный слой,
- `nn.ReLU`, `nn.LeakyReLU` — активации,
- `nn.Dropout` — dropout,
- `nn.BatchNorm1d` — BatchNorm для векторов/табличных данных.

Самый простой способ быстро собрать модель — `nn.Sequential`. Он принимает список слоёв и применяет их по очереди.

Пример (бинарная классификация):

```
import torch.nn as nn

model = nn.Sequential(
    nn.Linear(20, 64),
    nn.ReLU(),
    nn.Linear(64, 1)    # logits, sigmoid НЕ добавляем
)
```

Пример (многоклассовая классификация на k классов):

```
k = 10
model = nn.Sequential(
    nn.Linear(20, 128),
    nn.ReLU(),
    nn.Linear(128, k)   # logits, softmax НЕ добавляем
)
```

Замечание 15. Важно

`nn.Sequential` удобен, но он подходит не всегда. Если у вас есть ветвления, несколько входов или дополнительные выходы, тогда удобнее писать свой класс-наследник `nn.Module` (мы это сделаем дальше).

22.13.5 Loss-функции в PyTorch: как правильно

Бинарная классификация: используйте `BCEWithLogitsLoss`. Модель должна выдавать логит $z \in \mathbb{R}$, *без сигмоиды в модели.*

```
criterion = nn.BCEWithLogitsLoss()

logits = model(X_batch).squeeze(1)       # shape: [batch]
loss = criterion(logits, y_batch.float())
```

Многоклассовая классификация: используйте `CrossEntropyLoss`. Модель выдаёт logits размера $[batch, k]$, метки — целые классы $[batch]$.

```
criterion = nn.CrossEntropyLoss()

logits = model(X_batch)              # [batch, k]
loss = criterion(logits, y_batch)    # y_batch: LongTensor with class
 ↪  indices
```

Замечание 16. Частая ошибка

Не применяйте `softmax` перед `CrossEntropyLoss`: она ожидает logits и сама делает устойчивый `log_softmax`.

22.13.6 Оптимизатор и один шаг обучения

```
import torch.optim as optim

optimizer = optim.AdamW(model.parameters(), lr=1e-3,
 ↪  weight_decay=1e-4)

for X_batch, y_batch in train_loader:
    logits = model(X_batch)
    loss = criterion(logits, y_batch)

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

Gradient clipping (опционально):

```
torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
```

22.13.7 Dataset и DataLoader: как в sklearn, но батчами

```
from torch.utils.data import TensorDataset, DataLoader

dataset = TensorDataset(X_train_tensor, y_train_tensor)
train_loader = DataLoader(dataset, batch_size=64, shuffle=True)
```

Практика: делайте нормализацию как в sklearn, но внимательно:

- **fit** (считать среднее/стд) только на train,
- **transform** применять и к val/test.

22.13.8 Train/Eval режимы: Dropout и BatchNorm

Если в модели есть `Dropout` или `BatchNorm`, важно переключать режимы:

```
model.train()  # обучение (dropout включён, batchnorm по батчу)
...
model.eval()   # оценка (dropout выключен, batchnorm по running
 ↪  stats)
```

Оценку метрик делайте под `torch.no_grad()`.

22.13.9 Класс-наследник `nn.Module`: когда `Sequential` уже не хватает

```python
import torch
import torch.nn as nn

class MLP(nn.Module):
    def __init__(self, d_in, d_hidden, d_out):
        super().__init__()
        self.fc1 = nn.Linear(d_in, d_hidden)
        self.act = nn.ReLU()
        self.fc2 = nn.Linear(d_hidden, d_out)

    def forward(self, x):
        x = self.fc1(x)
        x = self.act(x)
        x = self.fc2(x)
        return x

model = MLP(d_in=20, d_hidden=64, d_out=1)
```

22.14. Регуляризация

Dropout: случайно зануляет часть нейронов во время обучения:

```
self.dropout = nn.Dropout(p=0.5)
```

Weight decay: реализуется через оптимизатор (особенно удобно в AdamW):

```
optimizer = optim.AdamW(model.parameters(), lr=1e-3,
 ↪  weight_decay=1e-4)
```

Early stopping: останавливаемся, если качество на валидации не улучшается.

```
best_val_loss = float("inf")
patience = 10
counter = 0

for epoch in range(num_epochs):
    model.train()
    for Xb, yb in train_loader:
        logits = model(Xb)
        loss = criterion(logits, yb)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    model.eval()
    with torch.no_grad():
        val_losses = []
        for Xb, yb in val_loader:
            logits = model(Xb)
            val_losses.append(criterion(logits, yb).item())
        val_loss = sum(val_losses) / len(val_losses)

    if val_loss < best_val_loss:
        best_val_loss = val_loss
        counter = 0
        torch.save(model.state_dict(), "best_model.pth")
    else:
        counter += 1
        if counter >= patience:
            break
```

22.15. Практические рекомендации (как собрать рабочий baseline)

1. **Предобработка данных:**
   - стандартизация числовых признаков (по train),
   - обработка пропусков,
   - аккуратное кодирование категориальных признаков.

2. **Архитектура (для табличных данных как старт):**
   - 2–3 скрытых слоя,
   - размеры, например: 128 $\rightarrow$ 64 $\rightarrow$ 32,
   - ReLU/Leaky ReLU,
   - (опционально) BatchNorm после Linear,
   - Dropout 0.1–0.5, если есть переобучение.
3. **Обучение:**
   - AdamW, $\eta \approx 10^{-3}$ как старт,
   - batch size 32–256,
   - валидация + early stopping,
   - при нестабильности: уменьшить LR, добавить clipping.
4. **Отладка:**
   - проверьте, что loss падает на маленьком подмножестве данных,
   - проверьте формы тензоров (`shape`) и типы (`dtype`),
   - если `NaN`: уменьшите LR, проверьте нормализацию, используйте устойчивые loss (`*WithLogits`).
5. **Метрики:**
   - бинарная классификация: Accuracy, ROC AUC, F1,
   - дисбаланс: Precision/Recall/F1, PR-AUC,
   - всегда смотрите confusion matrix.

23. Свёрточные нейронные сети

23.1. Введение

Сверточные нейронные сети (Convolutional Neural Networks, CNN) представляют собой класс архитектур глубокого обучения, специально разработанных для обработки данных с пространственной структурой, в первую очередь изображений. В отличие от полносвязных нейронных сетей, CNN используют операцию свертки для извлечения локальных признаков и разделение параметров (weight sharing), что радикально снижает число обучаемых параметров и позволяет эффективно работать с высокоразмерными данными.

23.2. Мотивация: почему полносвязные сети плохо подходят для изображений

23.2.1 Формат данных изображений

Цифровое цветное изображение естественно представляется в виде трехмерного тензора размерности $H \times W \times C$, где:

- H — высота изображения в пикселях;
- W — ширина изображения в пикселях;
- C — число цветовых каналов (обычно $C = 3$ для RGB).

Каждый элемент тензора $x_{i,j,c}$ принимает значения в диапазоне $[0, 255]$ для формата uint8 или нормализованные значения в $[0, 1]$ после предобработки.

Пример 23.1. Пример числа элементов

Пусть стандартное изображение имеет размер $224 \times 224 \times 3$. Общее число элементов составляет:

$$N = 224 \cdot 224 \cdot 3 = 150\,528.$$

23.2.2 Проблема числа параметров

Полносвязный слой (MLP) требует разворачивания многомерного тензора в вектор и умножения на матрицу весов. Пусть на вход подается изображение размера $H \times W \times C$, развернутое в вектор длины $N = HWC$, а на выходе слоя мы хотим получить M нейронов. Тогда матрица весов имеет размер $M \times N$, а общее число параметров равно:

$$\text{Параметров} = M \cdot N + M = M(N + 1).$$

Пример 23.2. Пример числа параметров

Пусть входное изображение $224 \times 224 \times 3$ ($N = 150\,528$), а первый скрытый слой содержит $M = 4096$ нейронов (как в классической архитектуре VGG). Тогда:

$$\text{Параметров} = 4096 \cdot 150\,528 + 4096 \approx 616\,562\,688.$$

Это более 600 миллионов параметров только в одном слое!

Такое количество параметров приводит к:

- огромным вычислительным затратам;

- высоким требованиям к памяти;
- риску переобучения даже на больших датасетах;
- длительному времени обучения.

23.2.3 Проблема инвариантности и локальности

Помимо количества параметров, полносвязные сети не учитывают два важнейших свойства изображений:

Локальность зависимостей. Пиксели изображения сильно коррелированы с пространственно близкими пикселями и почти независимы от удаленных. Кошка на фотографии состоит из локальных паттернов (уши, глаза, усы), а не из произвольных комбинаций пикселей со всего изображения. Полносвязный слой не учитывает эту структуру и "смешивает" все пиксели равноправно.

Отсутствие инвариантности к сдвигам. Если объект сдвинут на несколько пикселей влево, MLP воспринимает это как совершенно другое изображение: активируются другие веса, так как индексы пикселей изменились. Полносвязная сеть должна заново выучить, что это тот же объект, для каждой возможной позиции. Аналогично для поворотов и масштабирования.

23.2.4 Ключевые идеи CNN

Сверточные нейронные сети решают перечисленные проблемы за счет трех принципов:

1. **Локальные связи**: каждый нейрон связан только с небольшим локальным окном входа (receptive field), а не со всеми пикселями.
2. **Разделение параметров (weight sharing)**: один и тот же набор весов (ядро свертки) применяется ко всем позициям изображения. Это радикально снижает число параметров и обеспечивает трансляционную инвариантность.
3. **Иерархическое представление**: последовательность сверточных слоев извлекает признаки от простых (границы, углы) к сложным (текстуры, части объектов, целые объекты).

23.3. Операция свертки: интуиция и математика

23.3.1 Одномерная дискретная свертка

Начнем с одномерного случая для простоты.

Определение 23.1. Одномерная дискретная свертка

Дискретная свертка одномерных сигналов $x : \mathbb{Z} \to \mathbb{R}$ и $w : \mathbb{Z} \to \mathbb{R}$ определяется как:

$$(x * w)[n] = \sum_{k=-\infty}^{\infty} x[k]\, w[n-k], \quad n \in \mathbb{Z}.$$

На практике сигналы имеют конечную длину, и мы работаем с конечными суммами. Пусть $x \in \mathbb{R}^N$ и ядро $w \in \mathbb{R}^k$ (обычно $k \ll N$). Тогда:

$$y[n] = \sum_{m=0}^{k-1} x[n+m]\, w[m], \quad n = 0, \dots, N-k.$$

Замечание 17. Кросс-корреляция

В литературе по машинному обучению и во многих библиотеках (включая PyTorch) вместо свертки используется *кросс-корреляция*:

$$(x \star w)[n] = \sum_{m=0}^{k-1} x[n+m]\, w[m],$$

которая отличается от классической свертки отсутствием отражения ядра. С точки зрения обучения нейросети это не важно, так как веса ядра w оптимизируются, и сеть может выучить "отраженное" ядро, если это необходимо.

Пример 23.3. Одномерная свертка вручную

Пусть $x = [1, 2, 3, 4, 5]$ и ядро $w = [1, 0, -1]$. Вычислим кросс-корреляцию:

$$\begin{aligned} y[0] &= x[0]\cdot w[0] + x[1]\cdot w[1] + x[2]\cdot w[2] \\ &= 1\cdot 1 + 2\cdot 0 + 3\cdot(-1) = 1 + 0 - 3 = -2, \\ y[1] &= x[1]\cdot 1 + x[2]\cdot 0 + x[3]\cdot(-1) = 2 + 0 - 4 = -2, \\ y[2] &= x[2]\cdot 1 + x[3]\cdot 0 + x[4]\cdot(-1) = 3 + 0 - 5 = -2. \end{aligned}$$

Итого: $y = [-2, -2, -2]$.
Геометрический смысл: ядро $[1, 0, -1]$ вычисляет разность между соседними элементами с шагом 2 (дискретная производная).

23.3.2 Двумерная свертка для изображений

Для изображений естественно обобщить операцию на двумерный случай.

Определение 23.2. Двумерная сетка для изображений

Пусть $x \in \mathbb{R}^{H\times W}$ — входное изображение (одноканальное), $w \in \mathbb{R}^{k\times k}$ — ядро свертки. Двумерная кросс-корреляция определяется как:

$$y[i,j] = \sum_{u=0}^{k-1}\sum_{v=0}^{k-1} x[i+u, j+v]\, w[u,v], \quad i = 0, \dots, H-k,\ j = 0, \dots, W-k.$$

Здесь $y[i,j]$ — элемент выходной карты признаков (feature map), получаемый "наложением" ядра w на окно входа с левым верхним углом в позиции (i,j).

Пример 23.4. Двумерная свертка 3x3

Рассмотрим входное изображение 5×5 и ядро 3×3:

$$x = \begin{bmatrix} 1 & 2 & 3 & 0 & 1 \\ 4 & 5 & 6 & 2 & 3 \\ 7 & 8 & 9 & 4 & 5 \\ 0 & 1 & 2 & 6 & 7 \\ 3 & 4 & 5 & 8 & 9 \end{bmatrix}, \quad w = \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}.$$

Вычислим первый элемент выхода $y[0,0]$:

$$\begin{aligned} y[0,0] &= x[0,0]\cdot 1 + x[0,1]\cdot 0 + x[0,2]\cdot(-1) \\ &\quad + x[1,0]\cdot 1 + x[1,1]\cdot 0 + x[1,2]\cdot(-1) \\ &\quad + x[2,0]\cdot 1 + x[2,1]\cdot 0 + x[2,2]\cdot(-1) \\ &= 1 - 3 + 4 - 6 + 7 - 9 = -6. \end{aligned}$$

Аналогично вычисляются остальные элементы. Размер выхода: $(H-k+1)\times(W-k+1) = 3\times 3$.
Ядро w — это вертикальный *детектор границ Собеля*: оно реагирует на вертикальные перепады яркости.

23.3.3 Многоканальная свертка

Цветные изображения имеют несколько каналов (RGB).
Обобщим операцию на многоканальный случай.

Определение 23.3. Многоканальная свертка

Пусть входной тензор $x \in \mathbb{R}^{C_{in}\times H\times W}$ (формат PyTorch: channels first), а ядро $w \in \mathbb{R}^{C_{out}\times C_{in}\times k\times k}$. Тогда c-я выходная карта признаков вычисляется как:

$$y_c[i,j] = \sum_{d=1}^{C_{in}} \sum_{u=0}^{k-1} \sum_{v=0}^{k-1} x_d[i+u, j+v]\, w_{c,d}[u,v] + b_c,$$

где $c = 1, \dots, C_{out}$, b_c — смещение (bias) для c-го выходного канала.

Интерпретация:

- Каждое ядро w_c "просматривает" *все* входные каналы одновременно.
- Для каждого пространственного положения (i,j) ядро суммирует взвешенные значения по всем каналам и получает одно число $y_c[i,j]$.
- Набор из C_{out} различных ядер порождает C_{out} выходных карт признаков.

23.3.4 Подсчет параметров

Сверточный слой с параметрами (C_{in}, C_{out}, k) имеет:

$$\text{Параметров} = C_{out}\cdot C_{in}\cdot k^2 + C_{out},$$

где $C_{out}\cdot C_{in}\cdot k^2$ — веса ядер, C_{out} — смещения.

Пример 23.5. Сравнение CNN и MLP

Рассмотрим первый слой сети для изображений $224\times 224\times 3$.
Полносвязный слой: выход $M = 4096$ нейронов.

$$\text{Параметров}_{\text{FC}} = 150\,528\cdot 4096 + 4096 \approx 616\,562\,688.$$

Сверточный слой: ядра 7×7, выход $C_{out} = 64$ канала.

$$\text{Параметров}_{\text{Conv}} = 64\cdot 3\cdot 7^2 + 64 = 9\,408 + 64 = 9\,472.$$

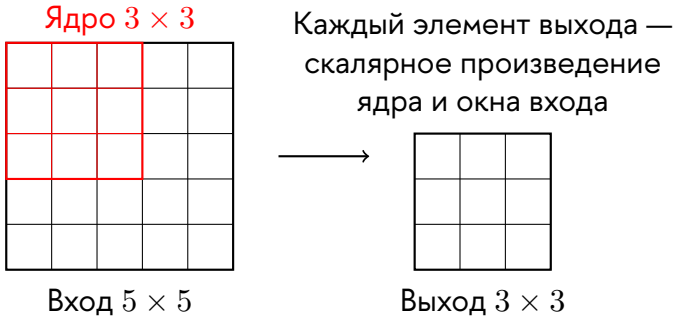
Сокращение числа параметров: в $\frac{616\,562\,688}{9\,472} \approx 65\,000$ раз!

Таблица 12: Сравнение числа параметров

Тип слоя	Вход	Выход	Параметров
Fully Connected	$224 \times 224 \times 3 = 150\,528$	4096	616,562,688
Convolutional (7×7)	$224 \times 224 \times 3$	$112 \times 112 \times 64$	9,472

23.3.5 Визуализация свертки

Операцию свертки можно визуализировать как "скользящее окно", которое перемещается по изображению и вычисляет скалярное произведение с ядром в каждой позиции.



23.4. Параметры сверточного слоя

В практических реализациях операция свертки управляется несколькими гиперпараметрами, которые влияют на размер выходной карты признаков, receptive field и вычислительную сложность.

23.4.1 Stride (шаг)

Определение 23.4. Stride

Stride s — это шаг, с которым ядро свертки перемещается по входу. При stride s ядро сдвигается на s пикселей по горизонтали и вертикали, а не на 1.

Формула размера выхода (без padding):

$$H_{out} = \left\lfloor \frac{H - k}{s} \right\rfloor + 1, \qquad W_{out} = \left\lfloor \frac{W - k}{s} \right\rfloor + 1.$$

Пример 23.6. Вычисление шага вручную

Вход 7×7, ядро 3×3, stride $s = 2$:

$$H_{out} = \left\lfloor \frac{7 - 3}{2} \right\rfloor + 1 = 2 + 1 = 3.$$

Выход: 3×3.
При stride $s = 1$: $H_{out} = 7 - 3 + 1 = 5$ (выход 5×5).

Зачем нужен stride > 1:

- уменьшает пространственный размер карт признаков (снижает вычислительные затраты);
- увеличивает receptive field последующих слоев;
- может заменять pooling-операции.

23.4.2 Padding (дополнение)

При свертке без padding размер выхода всегда меньше входа. Это приводит к двум проблемам:

1. пиксели на границах изображения участвуют в меньшем числе сверток, чем центральные (потеря информации);
2. после нескольких слоев пространственный размер сильно уменьшается.

Определение 23.5. Padding

Padding p — это добавление p строк/столбцов нулей по периметру входа перед применением свертки.

Формула размера выхода с padding:

$$H_{out} = \left\lfloor \frac{H + 2p - k}{s} \right\rfloor + 1, \qquad W_{out} = \left\lfloor \frac{W + 2p - k}{s} \right\rfloor + 1.$$

Типы padding:

- `valid` (без padding): $p = 0$, размер уменьшается.
- `same`: padding подбирается так, чтобы при stride $s = 1$ выход имел тот же размер, что и вход. Для ядра $k \times k$ нужно $p = \lfloor k/2 \rfloor$.

Пример 23.7. Сравнение размерностей

Вход 5×5, ядро 3×3, stride $s = 1$.
Без padding ($p = 0$):

$$H_{out} = \frac{5 - 3}{1} + 1 = 3.$$

Выход: 3×3.
С padding $p = 1$:

$$H_{out} = \frac{5 + 2 \cdot 1 - 3}{1} + 1 = \frac{7 - 3}{1} + 1 = 5.$$

Выход: 5×5 (размер сохранен).

23.4.3 Dilation (разреженная свертка)

Определение 23.6.

Dilation (или atrous convolution) d — это параметр, задающий шаг между элементами ядра. При $d = 1$ ядро применяется как обычно, при $d > 1$ между элементами ядра вставляются промежутки.

Разреженная свертка

Эффективный размер ядра с dilation:

$$k_{\text{eff}} = d(k-1)+1.$$

Формула размера выхода:

$$H_{out} = \left\lfloor \frac{H + 2p - k_{\text{eff}}}{s} \right\rfloor + 1.$$

Пример 23.8. Пример при $d = 2$

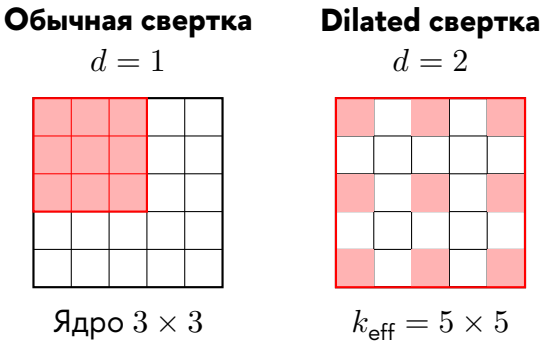
Ядро 3×3 с dilation $d = 2$:

$$k_{\text{eff}} = 2(3-1)+1 = 5.$$

Фактически ядро "просматривает" область 5×5, но использует только $3 \times 3 = 9$ весов.

Зачем нужен dilation:

- увеличивает receptive field без увеличения числа параметров;
- полезен в задачах, где важен глобальный контекст (например, сегментация).



23.4.4 Сводная таблица формул

Таблица 13: Формулы размера выхода сверточного слоя

Параметры	Формула H_{out}
Базовая (без padding, stride=1)	$H - k + 1$
С padding p	$H + 2p - k + 1$
С stride s	$\left\lfloor \frac{H+2p-k}{s} \right\rfloor + 1$
С dilation d	$\left\lfloor \frac{H+2p-k_{\text{eff}}}{s} \right\rfloor + 1$, где $k_{\text{eff}} = d(k-1)+1$

23.5. Backpropagation через сверточный слой

Для обучения CNN необходимо вычислять градиенты функции потерь по весам ядер и по входу. Рассмотрим сначала концептуальный подход, а затем строгий вывод через представление свертки в виде матричного умножения.

23.5.1 Концептуальное понимание

Рассмотрим сверточный слой как функцию $y = f(x; w)$, где x — вход, w — веса ядра, y — выход. Пусть L — функция потерь. Из цепного правила:

$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial w}, \qquad \frac{\partial L}{\partial x} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial x}.$$

Градиенты $\frac{\partial L}{\partial y}$ приходят с последующего слоя (это карта ошибок той же размерности, что и y). Нужно вычислить:

- $\frac{\partial L}{\partial w}$ — градиент по весам ядра (для обновления параметров);
- $\frac{\partial L}{\partial x}$ — градиент по входу (для распространения ошибки на предыдущий слой).

23.5.2 Представление свертки как матричного умножения

Для строгого вывода представим одномерную свертку в виде умножения на разреженную матрицу (Toeplitz-матрицу).

23.5.3 Одномерный случай

Пусть $x \in \mathbb{R}^N$ — входной вектор, $w \in \mathbb{R}^k$ — ядро. Свертка $y = x * w$ может быть записана как $y = Kx$, где K — матрица размера $(N - k + 1) \times N$, строки которой содержат сдвинутые копии ядра w.

Пример 23.9. Пример вычисления вручную при одномерном случае

Пусть $N = 5$, $k = 3$, $w = [w_0, w_1, w_2]$. Тогда:

$$K = \begin{bmatrix} w_0 & w_1 & w_2 & 0 & 0 \\ 0 & w_0 & w_1 & w_2 & 0 \\ 0 & 0 & w_0 & w_1 & w_2 \end{bmatrix}.$$

Умножение $y = Kx$ дает:

$$\begin{aligned} y[0] &= w_0 x[0] + w_1 x[1] + w_2 x[2], \\ y[1] &= w_0 x[1] + w_1 x[2] + w_2 x[3], \\ y[2] &= w_0 x[2] + w_1 x[3] + w_2 x[4], \end{aligned}$$

что совпадает с определением свертки.

23.5.4 Градиент по входу

Пусть L — скалярная функция потерь. Градиент по x:

$$\frac{\partial L}{\partial x} = K^\top \frac{\partial L}{\partial y}.$$

Транспонированная матрица $K^\top$ имеет размер $N \times (N - k + 1)$:

$$K^\top = \begin{bmatrix} w_0 & 0 & 0 \\ w_1 & w_0 & 0 \\ w_2 & w_1 & w_0 \\ 0 & w_2 & w_1 \\ 0 & 0 & w_2 \end{bmatrix}.$$

Умножение $K^\top \frac{\partial L}{\partial y}$ соответствует свертке вектора $\frac{\partial L}{\partial y}$ с *отраженным* ядром w с дополнением (padding).

Замечание 18. Градиент по входу

Градиент по входу — это "полная" свертка (full convolution) карты ошибок $\frac{\partial L}{\partial y}$ с отраженным ядром. В терминах операций:

$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial y} \circledast w_{\text{flip}},$$

где $\circledast$ — полная свертка (с padding), $w_{\text{flip}}[i] = w[k-1-i]$.

23.5.5 Градиент по весам

Градиент по весам ядра можно вывести, рассматривая каждый вес w_j отдельно:

$$\frac{\partial L}{\partial w_j} = \sum_n \frac{\partial L}{\partial y[n]} \frac{\partial y[n]}{\partial w_j}.$$

Из определения свертки:

$$y[n] = \sum_{m=0}^{k-1} x[n+m] w[m],$$

следует $\frac{\partial y[n]}{\partial w[j]} = x[n+j]$. Тогда:

$$\frac{\partial L}{\partial w[j]} = \sum_n \frac{\partial L}{\partial y[n]} x[n+j].$$

Это снова свертка: входного сигнала x с картой ошибок $\frac{\partial L}{\partial y}$.

Замечание 19. Градиент по весам

Градиент по весам — это свертка входа x с картой ошибок $\frac{\partial L}{\partial y}$:

$$\frac{\partial L}{\partial w} = x * \frac{\partial L}{\partial y}.$$

23.5.6 Двумерный случай

Для двумерной свертки выводы аналогичны:

- **Градиент по входу:** полная свертка карты ошибок $\frac{\partial L}{\partial y}$ с отраженным ядром w (rotation $180°$).
- **Градиент по весам:** свертка входной карты x с картой ошибок $\frac{\partial L}{\partial y}$.

Формально:

$$\frac{\partial L}{\partial x}[i,j] = \sum_{u=0}^{k-1} \sum_{v=0}^{k-1} \frac{\partial L}{\partial y}[i-u, j-v]\, w[u,v],$$

$$\frac{\partial L}{\partial w}[u,v] = \sum_i \sum_j \frac{\partial L}{\partial y}[i,j]\, x[i+u, j+v].$$

23.5.7 Численный пример

Рассмотрим простой случай: вход $x = [1, 2, 3, 4]$, ядро $w = [a, b]$ (два веса), выход $y = [y_0, y_1, y_2]$.

Forward pass:

$$y_0 = a \cdot 1 + b \cdot 2,$$
$$y_1 = a \cdot 2 + b \cdot 3,$$
$$y_2 = a \cdot 3 + b \cdot 4.$$

Пусть $L = (y_0 - t_0)^2 + (y_1 - t_1)^2 + (y_2 - t_2)^2$ (MSE loss). Градиенты по выходу:

$$\frac{\partial L}{\partial y_i} = 2(y_i - t_i) \equiv \delta_i.$$

Градиент по весам:

$$\frac{\partial L}{\partial a} = \delta_0 \cdot 1 + \delta_1 \cdot 2 + \delta_2 \cdot 3,$$
$$\frac{\partial L}{\partial b} = \delta_0 \cdot 2 + \delta_1 \cdot 3 + \delta_2 \cdot 4.$$

Градиент по входу (с padding):

$$\frac{\partial L}{\partial x[0]} = \delta_0 \cdot a,$$
$$\frac{\partial L}{\partial x[1]} = \delta_0 \cdot b + \delta_1 \cdot a,$$
$$\frac{\partial L}{\partial x[2]} = \delta_1 \cdot b + \delta_2 \cdot a,$$
$$\frac{\partial L}{\partial x[3]} = \delta_2 \cdot b.$$

Это соответствует полной свертке вектора $[\delta_0, \delta_1, \delta_2]$ с отраженным ядром $[b, a]$.

23.6. Pooling (Subsampling)

После сверточных слоев в CNN часто используются слои **pooling** (подвыборки), которые уменьшают пространственные размеры карт признаков. Это приводит к снижению числа параметров, вычислительных затрат и помогает бороться с переобучением.

23.6.1 Max Pooling

Определение 23.7. Max Pooling

Max pooling применяет операцию максимума к каждому неперекрывающемуся окну размера $p \times p$:

$$y[i, j] = \max_{0 \le u, v < p} x[i \cdot s + u, j \cdot s + v]$$

где s — stride (обычно $s = p$, неперекрывающиеся окна).

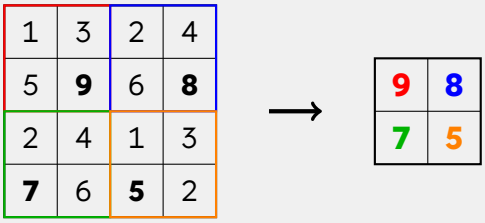
Формула размера выхода:

$$H_{out} = \left\lfloor \frac{H_{in} - p}{s} \right\rfloor + 1, \quad W_{out} = \left\lfloor \frac{W_{in} - p}{s} \right\rfloor + 1$$

Пример 23.10. Max Pooling 2×2

Рассмотрим входную карту признаков 4×4 и max pooling с окном 2×2, stride = 2:

Вход 4×4 **Выход** 2×2



Вычисления:

$$y[0,0] = \max(1,3,5,9) = 9$$
$$y[0,1] = \max(2,4,6,8) = 8$$
$$y[1,0] = \max(2,4,7,6) = 7$$
$$y[1,1] = \max(1,3,5,2) = 5$$

23.6.2 Average Pooling

Определение 23.8. Average Pooling

Average pooling вычисляет среднее значение в каждом окне:

$$y[i,j] = \frac{1}{p^2} \sum_{u=0}^{p-1} \sum_{v=0}^{p-1} x[i \cdot s + u, j \cdot s + v]$$

Для того же примера:

$$y[0,0] = \frac{1+3+5+9}{4} = 4.5, \quad y[0,1] = \frac{2+4+6+8}{4} = 5.0$$

23.6.3 Global Average Pooling (GAP)

Определение 23.9. Global Average Pooling

GAP усредняет всю карту признаков размера $H \times W$ до одного числа:

$$y_c = \frac{1}{H \cdot W} \sum_{i=0}^{H-1} \sum_{j=0}^{W-1} x_c[i,j]$$

для каждого канала c.

Применение: Часто используется перед финальным классификационным слоем вместо flatten + fully connected. Это радикально сокращает число параметров.

Пример: Вместо

$$7 \times 7 \times 512 \xrightarrow{\text{flatten}} 25{,}088 \xrightarrow{\text{FC}} 1000 \quad \text{(25 млн параметров)}$$

используем

$$7 \times 7 \times 512 \xrightarrow{\text{GAP}} 512 \xrightarrow{\text{FC}} 1000 \quad \text{(512 тыс. параметров)}$$

23.6.4 Backpropagation через Max Pooling

Градиент распространяется только через **максимальный элемент** в каждом окне.

Теорема 23.1. Градиент Max Pooling

Пусть $y[i,j] = \max_{(u,v)\in R_{ij}} x[u,v]$, где R_{ij} — окно pooling. Тогда:

$$\frac{\partial L}{\partial x[u,v]} = \begin{cases} \frac{\partial L}{\partial y[i,j]}, & \text{если } x[u,v] = \max_{R_{ij}} x \\ 0, & \text{иначе} \end{cases}$$

Интуиция: Только максимальный элемент "виноват"в значении выхода, остальные не влияют.

Пример 23.11. Backprop через Max Pooling

Вход 2×2:

$$X = \begin{bmatrix} 1 & 3 \\ 5 & 2 \end{bmatrix}$$

Max pooling 2×2 дает $y = 5$.
Если $\frac{\partial L}{\partial y} = 0.8$, то:

$$\frac{\partial L}{\partial X} = \begin{bmatrix} 0 & 0 \\ 0.8 & 0 \end{bmatrix}$$

Градиент идет только в позицию $(1,0)$, где был максимум.

23.6.5 Strided Convolution как альтернатива

Современные архитектуры иногда заменяют pooling на **свертку с stride > 1**:

$$\text{Conv}(k=3, s=2) \quad \text{вместо} \quad \text{Conv}(k=3, s=1) + \text{MaxPool}(2 \times 2)$$

Преимущества:

- Обучаемые параметры (веса свертки) вместо фиксированной операции max
- Более гибкое обучение downsampling

Недостатки:

- Больше параметров и вычислений

Таблица 14: Сравнение методов подвыборки

Метод	Параметров	Обучается	Применение
Max Pooling	0	Нет	Выделение ярких признаков
Average Pooling	0	Нет	Сглаживание
Global Average Pooling	0	Нет	Перед классификатором
Strided Conv (3×3, $s=2$)	$9C_{in}C_{out}$	Да	Современные архитектуры

23.7. Receptive Field (Рецептивное поле)

Определение 23.10. Receptive Field

Рецептивное поле (receptive field, RF) нейрона в слое l — это область входного изображения, которая влияет на активацию этого нейрона.

Геометрический смысл: RF показывает, какую область "видит"конкретный нейрон. Чем глубже слой, тем больше RF.

Для цепочки сверточных слоев с ядрами $k \times k$ и stride = 1:

Теорема 23.2. Рост Receptive Field

Если все свертки имеют ядро $k \times k$ и stride = 1, то RF после L слоев:

$$RF_L = 1 + L \cdot (k-1)$$

База: RF первого слоя $RF_1 = k$.
Шаг: Пусть RF_l известен. После добавления слоя с ядром $k \times k$:

- Каждый нейрон видит $k \times k$ окно в предыдущем слое
- Каждый элемент предыдущего слоя покрывает RF_l на входе
- Крайние элементы окна $k \times k$ расширяют RF на $(k-1)$ с каждой стороны

Итого:

$$RF_{l+1} = RF_l + (k-1)$$

Раскрывая рекуррентность:

$$RF_L = k + (k-1) + (k-1) + \cdots + (k-1) = k + (L-1)(k-1) = 1 + L(k-1)$$

Пример 23.12. Цепочка из трех сверток 3×3

$$RF_1 = 3, \quad RF_2 = 3 + 2 = 5, \quad RF_3 = 5 + 2 = 7$$

По формуле: $RF_3 = 1 + 3 \cdot (3-1) = 7$

23.7.1 RF для сверток с разными параметрами

Более общая формула для stride s и dilation d:

Теорема 23.3. Общая формула RF

Для слоя l с параметрами (k_l, s_l, d_l):

$$RF_l = RF_{l-1} + (k_{\text{eff},l} - 1) \cdot \prod_{i=1}^{l-1} s_i$$

где $k_{\text{eff}} = d(k-1) + 1$ — эффективный размер ядра.

Таблица 15: Рост Receptive Field для различных архитектур

Слой	Операция	RF	Размер карты	Комментарий
Вход	—	1	224×224	Исходное изображение
Conv1	3×3, $s = 1$	3	222×222	Базовая свертка
Conv2	3×3, $s = 1$	5	220×220	+2 к RF
Conv3	3×3, $s = 1$	7	218×218	+2 к RF
Pool	2×2, $s = 2$	8	109×109	Pooling удваивает шаг
Conv4	3×3, $s = 1$	12	107×107	RF растет быстрее

23.7.2 Почему важен большой RF?

Контекст: Большой RF позволяет нейронам учитывать широкий контекст:

- Распознавание объектов требует видеть весь объект
- Семантическая сегментация нуждается в глобальном контексте
- Малый RF видит только текстуры, не объекты

Способы увеличения RF:

1. Добавить больше слоев: $RF = 1 + L(k-1)$
2. Использовать большие ядра: $k = 5$ или 7 вместо 3
3. Dilated convolutions: эффективный размер $k_{\text{eff}} = d(k-1) + 1$
4. Pooling: увеличивает эффективный шаг

23.8. Регуляризация в сверточных сетях

Глубокие CNN склонны к переобучению из-за большого числа параметров. Рассмотрим основные методы регуляризации.

23.8.1 Data Augmentation (Аугментация данных)

Идея:
Искусственно расширить обучающую выборку, применяя трансформации к изображениям.

Геометрические трансформации

1. **Horizontal Flip** (горизонтальное отражение):

$$x'[i, j] = x[i, W - j - 1]$$

2. **Random Crop** (случайная обрезка):
   - Из изображения $H \times W$ вырезаем $h \times w$, где $h < H$
   - Позиция случайна
3. **Rotation** (поворот на угол θ):

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

4. **Scale** (масштабирование) с коэффициентом $s \in [0.8, 1.2]$
5. **Translation** (сдвиг) на $(\Delta x, \Delta y)$

Цветовые трансформации

1. **Brightness** (яркость):
$$x'_{RGB} = x_{RGB} + \delta, \quad \delta \sim \mathcal{U}(-30, 30)$$
2. **Contrast** (контраст):
$$x'_{RGB} = \alpha \cdot x_{RGB}, \quad \alpha \sim \mathcal{U}(0.7, 1.3)$$
3. **Saturation** (насыщенность в HSV):
$$S' = S \cdot \beta, \quad \beta \sim \mathcal{U}(0.5, 1.5)$$
4. **Hue Shift** (сдвиг оттенка):
$$H' = (H + \Delta H) \mod 360$$

Cutout и Dropout-like методы

Определение 23.11. Cutout

Случайным образом маскируем квадратную область изображения размера $l \times l$, заполняя ее нулями или средним значением.

Формально:
Выбираем случайную позицию (i_0, j_0) и применяем:
$$x'[i,j] = \begin{cases} 0, & \text{если } i_0 \le i < i_0 + l \text{ и } j_0 \le j < j_0 + l \\ x[i,j], & \text{иначе} \end{cases}$$

Эффект:
Сеть учится распознавать объекты по частичной информации, не полагаясь на одну область.

23.8.2 Dropout в CNN

Определение 23.12. Dropout

Во время обучения случайно "выключаем"нейроны с вероятностью p:
$$h'_i = \begin{cases} 0, & \text{с вероятностью } p \\ \frac{h_i}{1-p}, & \text{с вероятностью } 1-p \end{cases}$$

Где применять в CNN:

- Не в сверточных слоях. Spatial correlations делают dropout неэффективным
- Только в полносвязных слоях (если они есть)
- Типично: $p = 0.5$ для FC, $p = 0.2$ для входного слоя

Альтернатива:
Spatial Dropout — выключаем целые каналы, а не отдельные пиксели.

23.8.3 Batch Normalization

Определение 23.13. Batch Normalization

Нормализуем активации по mini-batch:

$$\hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}, \quad y_i = \gamma \hat{x}_i + \beta$$

где μ_B, σ_B^2 — среднее и дисперсия в батче, γ, β — обучаемые параметры.

Место в CNN:

$$\text{Conv} \to \text{BatchNorm} \to \text{ReLU}$$

Эффекты:

- Стабилизирует обучение, позволяет использовать большие learning rates
- Регуляризирует за счет шума в μ_B, σ_B^2 (зависят от случайного батча)
- Уменьшает зависимость от инициализации

23.8.4 Label Smoothing

Определение 23.14. Label Smoothing

Заменяем "жесткие"метки на "мягкие":

$$y_i' = \begin{cases} 1 - \alpha + \frac{\alpha}{K}, & \text{если } i = y_{\text{true}} \\ \frac{\alpha}{K}, & \text{иначе} \end{cases}$$

где K — число классов, $\alpha \in [0.1, 0.2]$ — параметр сглаживания.

Пример: Для $K = 5$, истинный класс 2, $\alpha = 0.1$:

$$y = [0, 0, 1, 0, 0] \quad \to \quad y' = [0.02, 0.02, 0.92, 0.02, 0.02]$$

Эффект: Предотвращает overconfidence модели, улучшает калибровку вероятностей.

23.8.5 Mixup

Определение 23.15. Mixup

Создаем "смешанные"примеры:

$$\tilde{x} = \lambda x_i + (1 - \lambda) x_j, \quad \tilde{y} = \lambda y_i + (1 - \lambda) y_j$$

где $\lambda \sim \text{Beta}(\alpha, \alpha)$, обычно $\alpha = 0.2$.

Интерпретация: Модель обучается на линейных комбинациях примеров, что делает её более устойчивой.

Пример 23.13. Mixup

Смешиваем изображение кошки (класс 0) и собаки (класс 1) с $\lambda = 0.7$:

$$\tilde{x} = 0.7 \cdot x_{\text{cat}} + 0.3 \cdot x_{\text{dog}}$$

$$\tilde{y} = [0.7, 0.3]$$

Модель обучается предсказывать 70% кошка, 30% собака.

23.8.6 CutMix

Определение 23.16. CutMix

Комбинация Mixup и Cutout: вырезаем прямоугольную область из одного изображения и вставляем в другое.

$$\tilde{x}[i,j] = \begin{cases} x_A[i,j], & \text{если } (i,j) \in R \\ x_B[i,j], & \text{иначе} \end{cases}$$

$$\tilde{y} = \lambda y_A + (1-\lambda) y_B, \quad \lambda = \frac{|R|}{H \cdot W}$$

где R — прямоугольная область, $|R|$ — её площадь.

Таблица 16: Сравнение методов регуляризации в CNN

Метод	Стадия	Сложность	Эффект
Data Augmentation	Препроцессинг	Низкая	+2-5% accuracy
Dropout (FC)	Обучение	Низкая	Борьба с overfitting
Batch Normalization	Обучение	Средняя	Стабилизация, +скорость
Label Smoothing	Обучение	Низкая	Калибровка вероятностей
Mixup	Обучение	Средняя	+1-2% accuracy
CutMix	Обучение	Средняя	+1-3% accuracy

Практические рекомендации:

1. Всегда используйте Data Augmentation (flip, crop, color jitter)
2. Batch Normalization после каждой свертки
3. Dropout только в FC слоях (если есть)
4. Label Smoothing с $\alpha = 0.1$ для классификации
5. Mixup или CutMix для дополнительной регуляризации

23.9. Эволюция архитектур сверточных сетей

23.9.1 LeNet-5 (1998)

Определение 23.17. LeNet-5

Одна из первых успешных CNN, разработанная Яном Лекуном для распознавания рукописных цифр на чеках.

Архитектура LeNet-5:

$$\text{Input}(32 \times 32) \to \text{Conv1}(5 \times 5, 6) \to \text{AvgPool}(2 \times 2) \to \text{Conv2}(5 \times 5, 16) \to \text{AvgPool}(2 \times 2) \to \text{FC}(120) \to \text{FC}(84) \to \text{Output}(10)$$

Ключевые особенности:

- Использование **tanh** активации вместо ReLU (которой тогда не было)
- **Average pooling** вместо max pooling
- Малая глубина (7 слоев)
- Всего около 60,000 параметров

Пример 23.14. Расчет параметров LeNet-5

Conv1: $(5 \times 5 \times 1 + 1) \times 6 = 156$ параметров
Conv2: $(5 \times 5 \times 6 + 1) \times 16 = 2{,}416$ параметров
FC1: $(5 \times 5 \times 16) \times 120 + 120 = 48{,}120$ параметров
FC2: $120 \times 84 + 84 = 10{,}164$ параметра
Output: $84 \times 10 + 10 = 850$ параметров
Итого: $156 + 2{,}416 + 48{,}120 + 10{,}164 + 850 = 61{,}706$ параметров

23.9.2 AlexNet (2012)

Определение 23.18. AlexNet

Глубокая CNN, выигравшая ImageNet ILSVRC 2012 с огромным отрывом (top-5 error 15.3% vs 26.2% у второго места), ознаменовавшая начало эры глубокого обучения.

Архитектура AlexNet:

$$\begin{aligned}&\text{Input}(224 \times 224 \times 3)\\ &\to \text{Conv1}(11 \times 11, 96, s = 4) \to \text{MaxPool}(3 \times 3, s = 2)\\ &\to \text{Conv2}(5 \times 5, 256) \to \text{MaxPool}(3 \times 3, s = 2)\\ &\to \text{Conv3}(3 \times 3, 384) \to \text{Conv4}(3 \times 3, 384) \to \text{Conv5}(3 \times 3, 256) \to \text{MaxPool}(3 \times 3, s = 2)\\ &\to \text{FC}(4096) \to \text{Dropout}(0.5) \to \text{FC}(4096) \to \text{Dropout}(0.5) \to \text{FC}(1000)\end{aligned}$$

Революционные нововведения:

1. ReLU активация: $f(x) = \max(0, x)$ — быстрее обучается, чем tanh
2. GPU ускорение: обучение на 2 GPU (GTX 580)
3. Dropout в полносвязных слоях
4. Data Augmentation: random crops, horizontal flips, color jitter
5. Max Pooling вместо average pooling
6. Local Response Normalization (позже заменена на Batch Norm)

Число параметров: ≈ 60 млн (в основном в FC слоях)

23.9.3 VGG (2014)

Определение 23.19. VGG

Семейство архитектур (VGG-16, VGG-19), доказавших, что **глубина важнее** размера фильтров. Основная идея: заменить большие фильтры на стек маленьких 3×3.

Ключевая идея VGG:

Теорема 23.4. Декомпозиция фильтров

Два последовательных слоя 3×3 имеют тот же receptive field, что один слой 5×5, но:

- Меньше параметров: $2 \times (9C^2) = 18C^2$ vs $25C^2$
- Больше нелинейностей (2 ReLU вместо 1)
- Более выразительная функция

Доказательство.
Receptive field двух сверток 3×3:

$$RF_2 = 3 + (3 - 1) = 5$$

Параметры для C каналов:

- Два слоя 3×3: $C \times 3 \times 3 \times C + C \times 3 \times 3 \times C = 18C^2$
- Один слой 5×5: $C \times 5 \times 5 \times C = 25C^2$

Экономия: $(25 - 18)/25 = 28\%$ ■

Архитектура VGG-16:

Таблица 17: Структура VGG-16

Блок	Слои	Каналы	Размер карты
Input	—	3	224×224
Block 1	Conv 3×3 (×2) + MaxPool	64	112×112
Block 2	Conv 3×3 (×2) + MaxPool	128	56×56
Block 3	Conv 3×3 (×3) + MaxPool	256	28×28
Block 4	Conv 3×3 (×3) + MaxPool	512	14×14
Block 5	Conv 3×3 (×3) + MaxPool	512	7×7
FC	FC(4096) + FC(4096) + FC(1000)	—	—

Особенности:

- Единообразие: только 3×3 свертки и 2×2 pooling
- 16 слоев с весами (13 conv + 3 FC)
- ≈ 138 млн параметров (большинство в FC)
- Простота и элегантность

23.9.4 GoogLeNet / Inception (2014)

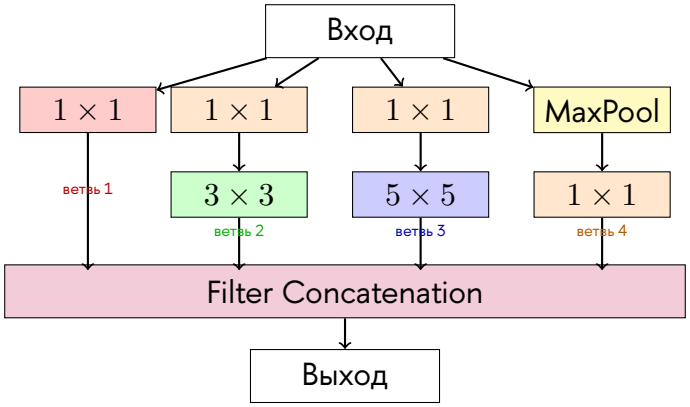
Определение 23.20. Inception модуль

Параллельная обработка с фильтрами разных размеров (1×1, 3×3, 5×5) и pooling, с последующей конкатенацией по каналам.

Мотивация: Разные объекты имеют разные масштабы. Inception позволяет сети **самой выбрать** оптимальный размер фильтра для каждого региона.

Структура Inception модуля:

Структура Inception модуля



Роль 1×1 сверток:

- Dimensionality reduction: $H \times W \times 256 \to H \times W \times 64$
- Сокращение вычислений перед дорогими 3×3 и 5×5
- Добавление нелинейности

Пример 23.15. Экономия параметров с 1×1

Без 1×1 редукции: $256 \times 3 \times 3 \times 256 = 589{,}824$ параметра
С редукцией $256 \to 64 \to 256$:

$$256 \times 1 \times 1 \times 64 + 64 \times 3 \times 3 \times 256 = 16{,}384 + 147{,}456 = 163{,}840$$

Экономия: $(589{,}824 - 163{,}840)/589{,}824 = 72\%$

23.10. Остаточные сети (ResNet)

23.10.1 Проблема затухающих градиентов

При увеличении глубины сети возникает **degradation problem**: точность сначала растет, затем **насыщается и падает** (не из-за overfitting!).

Замечание 20. Парадокс глубины

Теоретически, более глубокая сеть не должна быть хуже мелкой: она может просто скопировать веса мелкой и сделать остальные слои identity-отображениями. Но на практике этого не происходит из-за сложности оптимизации.

Причины деградации:

- Vanishing gradients: $\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial y} \prod_{i=1}^{L} \frac{\partial h_i}{\partial h_{i-1}}$ — произведение может стремиться к 0
- Сложность обучения identity mapping через цепочку нелинейностей

23.10.2 Skip Connection и Residual Block

Определение 23.21. Skip Connection (Residual Connection)

Прямое соединение входа блока с его выходом:

$$y = F(x, \{W_i\}) + x$$

где $F(x)$ — остаточная функция (residual), которую нужно обучить.

Ключевая идея ResNet: Вместо обучения желаемого отображения $H(x)$, обучаем **остаток**:

$$F(x) = H(x) - x$$

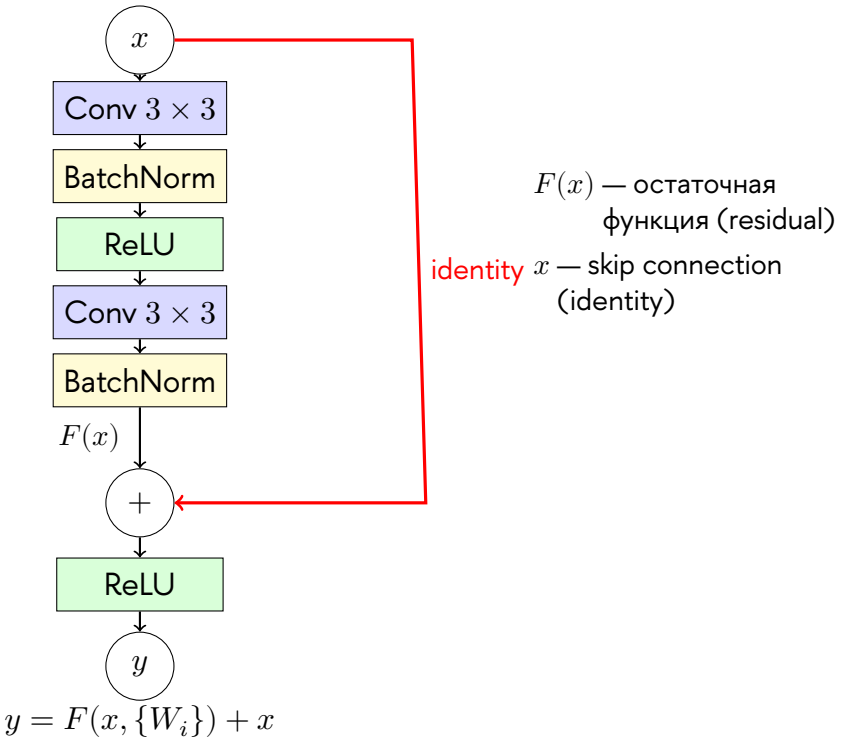
Итоговое отображение:

$$H(x) = F(x) + x$$

Почему это помогает?

1. Если identity — оптимально, сети легче обучить $F(x) = 0$, чем identity через нелинейности
2. Градиент проходит напрямую: $\frac{\partial}{\partial x}(F(x) + x) = \frac{\partial F}{\partial x} + 1$ — всегда есть слагаемое 1
3. Облегчает обучение очень глубоких сетей (152+ слоя)

Residual Block



23.10.3 Backpropagation через Skip Connection

Теорема 23.5. Градиент через Residual Block

Для $y = F(x) + x$:

$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial y}\left(\frac{\partial F}{\partial x} + 1\right)$$

Следствие: Даже если $\frac{\partial F}{\partial x} \to 0$ (vanishing gradient в F), градиент всегда содержит слагаемое $\frac{\partial L}{\partial y}$, которое "пробрасывается"напрямую.

Для цепочки из L residual блоков:

$$\frac{\partial L}{\partial x_0} = \frac{\partial L}{\partial x_L}\left(1 + \sum_{i=1}^{L} \frac{\partial F_i}{\partial x_{i-1}}\right)$$

Градиент не может занулиться полностью!

23.10.4 Архитектура ResNet

Basic Block (ResNet-18, ResNet-34):

Conv $3 \times 3, C$
BatchNorm, ReLU
Conv $3 \times 3, C$
BatchNorm
Add skip connection, ReLU

Bottleneck Block (ResNet-50, ResNet-101, ResNet-152):

Conv $1 \times 1, C/4$ (dimension reduction)
BatchNorm, ReLU
Conv $3 \times 3, C/4$
BatchNorm, ReLU
Conv $1 \times 1, C$ (dimension restoration)
BatchNorm
Add skip connection, ReLU

Зачем Bottleneck? Сокращение параметров:

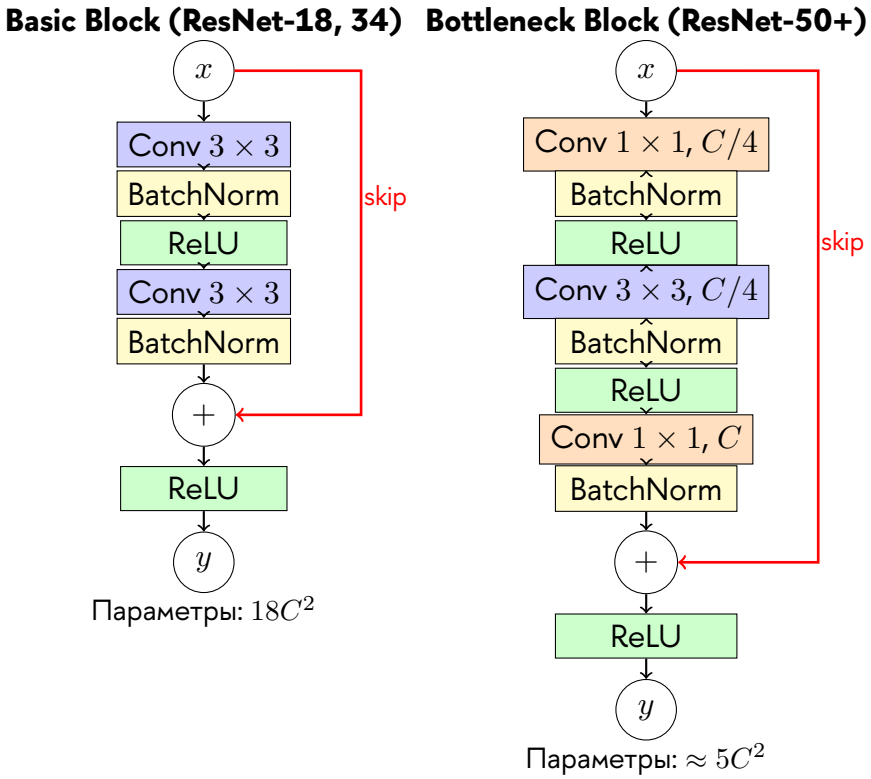
- Basic: $2 \times (3 \times 3 \times C \times C) = 18C^2$
- Bottleneck: $C \times \frac{C}{4} + \frac{C}{4} \times 3 \times 3 \times \frac{C}{4} + \frac{C}{4} \times C = \frac{C^2}{4} + \frac{9C^2}{16} + \frac{C^2}{4} \approx 5.1C^2$

Экономия: $\approx 72\%$

Таблица 18: Семейство ResNet архитектур

Модель	Слои	Блок	Параметров	Top-5 Error
ResNet-18	18	Basic	11.7M	10.2%
ResNet-34	34	Basic	21.8M	8.6%
ResNet-50	50	Bottleneck	25.6M	7.1%
ResNet-101	101	Bottleneck	44.6M	6.5%
ResNet-152	152	Bottleneck	60.2M	6.2%

23.10.5 Basic Block и Bottleneck визуально

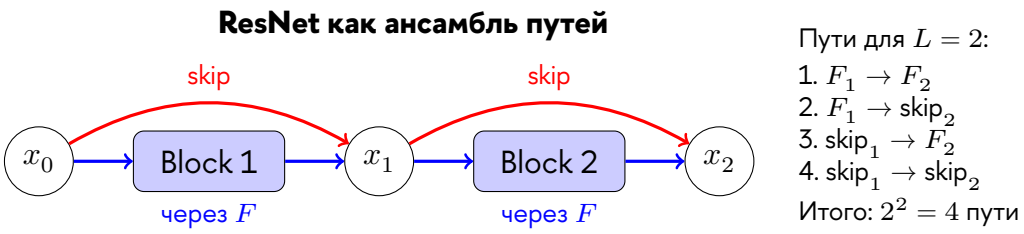


23.10.6 Интерпретации ResNet

1. Ансамбль путей

ResNet можно интерпретировать как **ансамбль экспоненциального числа путей** разной длины.

При L residual блоках существует 2^L различных путей от входа к выходу (каждый блок: использовать F или skip).



Для L блоков существует 2^L различных путей от входа к выходу

2. Связь с Dropout

Skip connections создают эффект, похожий на Dropout: разные пути активны, сеть более устойчива.

3. Gradient highway

Skip connections создают "магистраль"для прямого прохождения градиента, борясь с vanishing gradients.

Замечание 21. ResNet и Бустинг

ResNet можно рассматривать как аддитивную модель:

$$H(x) = x + F_1(x) + F_2(F_1(x) + x) + \cdots$$

где каждый блок добавляет "коррекцию"к текущему представлению — аналог gradient boosting.

23.10.7 Практическая реализация ResNet блоков

```python
import torch
import torch.nn as nn

class BasicBlock(nn.Module):
    def __init__(self, in_channels, out_channels, stride=1):
        super(BasicBlock, self).__init__()
        self.conv1 = nn.Conv2d(in_channels, out_channels,
                               kernel_size=3, stride=stride,
                               padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(out_channels)
        self.relu = nn.ReLU(inplace=True)
        self.conv2 = nn.Conv2d(out_channels, out_channels,
                               kernel_size=3, stride=1,
                               padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(out_channels)

        # Skip connection (если размерности не совпадают)
        self.shortcut = nn.Sequential()
        if stride != 1 or in_channels != out_channels:
            self.shortcut = nn.Sequential(
                nn.Conv2d(in_channels, out_channels,
                          kernel_size=1, stride=stride, bias=False),
                nn.BatchNorm2d(out_channels)
            )

    def forward(self, x):
        identity = self.shortcut(x)

        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)

        out = self.conv2(out)
        out = self.bn2(out)

```

```
        out += identity  # Skip connection!
        out = self.relu(out)

        return out
```

Ключевые моменты реализации:

1. BatchNorm идет **после** свертки, **перед** ReLU
2. Skip connection добавляется **перед финальным ReLU**
3. Если размерности не совпадают (stride $\neq 1$), нужна проекция через Conv 1×1
4. `bias=False` в Conv, так как BatchNorm имеет свой bias

23.10.8 Полная архитектура ResNet-18

Таблица 19: Детальная структура ResNet-18

Слой	Выход	Параметры	Слоев с весами
Conv1	$112 \times 112 \times 64$	7×7, stride=2	1
MaxPool	$56 \times 56 \times 64$	3×3, stride=2	0
Layer1	$56 \times 56 \times 64$	2× Basic Block (64)	4
Layer2	$28 \times 28 \times 128$	2× Basic Block (128)	4
Layer3	$14 \times 14 \times 256$	2× Basic Block (256)	4
Layer4	$7 \times 7 \times 512$	2× Basic Block (512)	4
AvgPool	$1 \times 1 \times 512$	Global Average Pool	0
FC	1000	Fully Connected	1
		Итого слоев:	**18**

Пояснение: Basic Block содержит 2 свертки 3×3. Поэтому:

$$1 \text{ (Conv1)} + 2 \times 2 \times 4 \text{ (Layers)} + 1 \text{ (FC)} = 1 + 16 + 1 = 18$$

Общее число слоев: $1 + 2 \times 8 + 1 = 18$ (отсюда название ResNet-18)

23.11. Transfer Learning (Перенос обучения)

23.11.1 Концепция

Определение 23.22. Transfer Learning

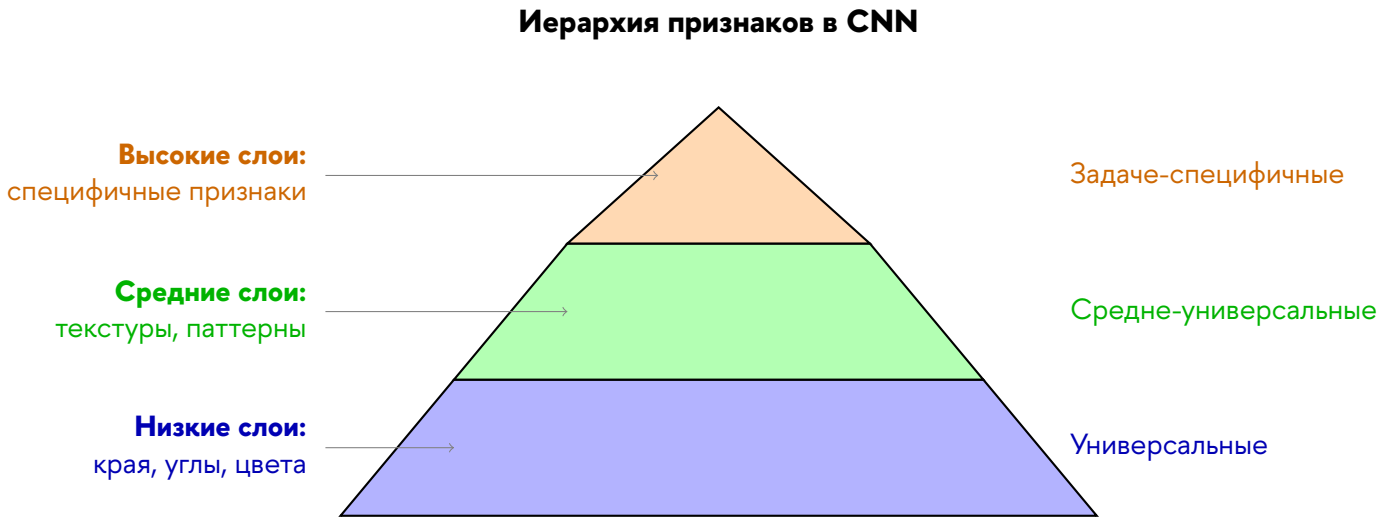
Использование модели, предобученной на большом датасете (например, ImageNet), в качестве отправной точки для новой задачи с меньшим количеством данных.

Мотивация:

- Обучение с нуля на ImageNet: недели на GPU кластере
- Малые датасеты: overfitting при обучении глубоких сетей
- Низкоуровневые признаки (края, текстуры) универсальны для всех изображений

Иерархия признаков в CNN:

- **Низкие слои**: универсальные признаки (края, углы, цвета)
- **Средние слои**: текстуры, паттерны
- **Высокие слои**: специфичные для задачи (части объектов, сцены)



23.11.2 Стратегии Transfer Learning

Стратегия 1: Feature Extractor (Замораживание)

1. Загружаем предобученную модель (ResNet-50 на ImageNet)
2. **Замораживаем** все слои: веса не обновляются
3. Заменяем последний FC слой на новый (для K классов вашей задачи)
4. Обучаем **только новый классификатор**

Когда использовать:

- Очень мало данных (< 1000 изображений)
- Данные похожи на ImageNet

Стратегия 2: Fine-Tuning (Дообучение)

1. Загружаем предобученную модель
2. Заменяем последний слой
3. Обучаем новый классификатор с замороженной базой
4. **Размораживаем** несколько последних слоев
5. Дообучаем с **маленьким learning rate** (например, 10^{-4})

Когда использовать:

- Средний размер датасета (1000-100k)

- Данные умеренно отличаются от ImageNet

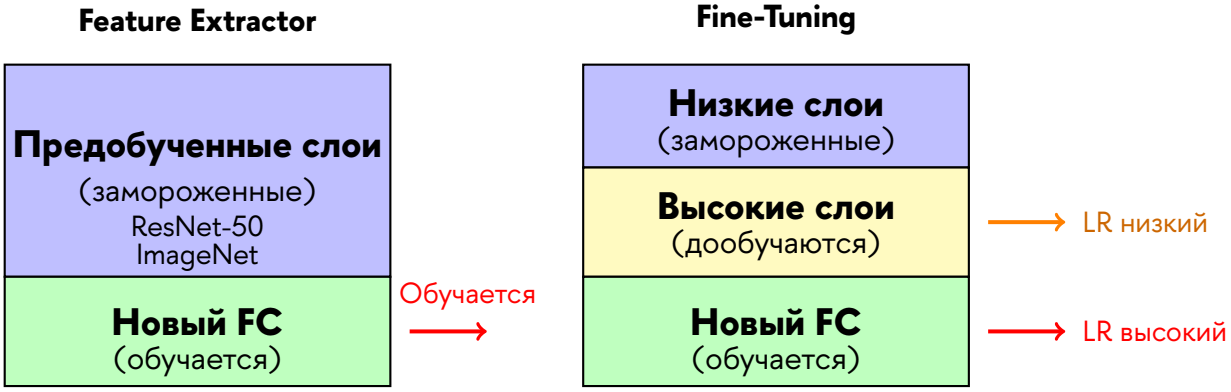


Таблица 20: Выбор стратегии Transfer Learning

Размер данных	Похожи на ImageNet	Отличаются от ImageNet
Малый (< 1k)	Feature extractor	Feature extractor + data aug
Средний (1k-100k)	Fine-tune последние слои	Fine-tune все слои
Большой (> 100k)	Fine-tune все	Обучение с нуля

23.11.3 Практическая реализация

```python
import torch
import torch.nn as nn
import torchvision.models as models

# Загружаем предобученный ResNet-50
model = models.resnet50(pretrained=True)

# Стратегия 1: Feature Extractor
# Замораживаем все слои
for param in model.parameters():
    param.requires_grad = False

# Заменяем последний слой (1000 -> 10 классов)
num_features = model.fc.in_features
model.fc = nn.Linear(num_features, 10)

# Обучаем только fc слой
optimizer = torch.optim.Adam(model.fc.parameters(), lr=1e-3)

# Стратегия 2: Fine-Tuning
# Размораживаем последние 2 блока (layer4)
for param in model.layer4.parameters():
    param.requires_grad = True

# Обучаем с меньшим LR для предобученных слоев
optimizer = torch.optim.Adam([
    {'params': model.fc.parameters(), 'lr': 1e-3},
    {'params': model.layer4.parameters(), 'lr': 1e-4}
])
```

Рекомендации:

- Learning rate для fine-tuning: в 10-100 раз меньше, чем при обучении с нуля
- Differential learning rates: больший LR для новых слоев, меньший для предобученных
- Data augmentation критически важна для малых датасетов
- Batch Normalization: использовать `model.eval()` если не дообучаем

23.12. U-Net и семантическая сегментация

23.12.1 Семантическая сегментация

Определение 23.23. Семантическая сегментация

Задача присвоения каждому пикселю изображения метки класса (pixel-wise classification).

Отличие от классификации:

- Классификация: один выход на изображение
- Сегментация: выход размера $H \times W \times K$ (K — число классов)

Примеры применений:

- Медицинская визуализация: сегментация органов, опухолей
- Автономные автомобили: дорога, машины, пешеходы
- Спутниковые снимки: здания, деревья, водоемы

23.12.2 Архитектура U-Net

Определение 23.24. U-Net

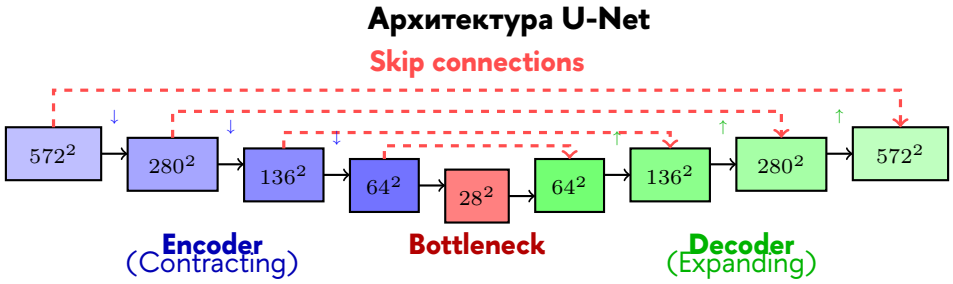
Encoder-Decoder архитектура с **skip connections** между соответствующими уровнями энкодера и декодера, образующая U-образную форму.

Структура U-Net:

1. **Encoder (Contracting Path)**:
   - Последовательность: Conv 3×3 ReLU Conv 3×3 ReLU MaxPool 2×2
   - Каждый downsampling удваивает число каналов
   - Путь: $572 \times 572 \times 1 \to 28 \times 28 \times 1024$
2. **Bottleneck**:
   - Самое узкое место: наименьший spatial размер, максимум каналов
3. **Decoder (Expanding Path)**:
   - Upsampling: **ConvTranspose2d** (transpose convolution) 2×2
   - **Конкатенация** с соответствующей картой из энкодера (skip connection)

- Conv $3 \times 3 \to$ ReLU $\to$ Conv $3 \times 3 \to$ ReLU
- Каждый upsampling вдвое уменьшает число каналов

4. **Выходной слой**:
   - Conv 1×1 для проекции на K классов
   - Softmax для вероятностей

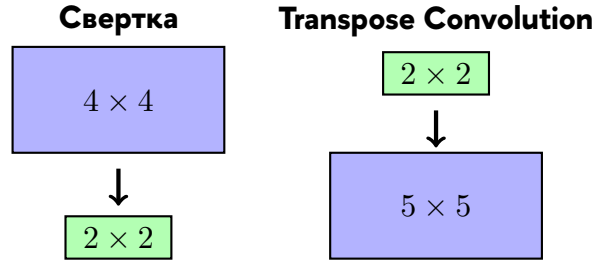


23.12.3 Transpose Convolution

Определение 23.25. Transpose Convolution (Deconvolution)

Операция, увеличивающая пространственный размер карты признаков. Формально: транспонированная версия прямой свертки.

Для обычной свертки: $y = Kx$, где K — Toeplitz-матрица.
Transpose convolution: $\tilde{x} = K^T y$ — восстанавливает размер.



Обычная: $H_{out} = \lfloor \frac{H_{in}-k}{s} \rfloor + 1$
Transpose: $H_{out} = (H_{in} - 1) \cdot s + k$

Пример 23.16. Transpose Convolution

Вход: 2×2, ядро 3×3, stride = 2, padding = 0
Выход: 5×5 (upsampling в 2.5 раза)
Формула:

$$H_{out} = (H_{in} - 1) \cdot s - 2p + k$$

Для нашего случая: $(2-1) \cdot 2 - 0 + 3 = 5$

Почему skip connections в U-Net?

- Encoder теряет пространственную информацию при downsampling
- Decoder восстанавливает размер, но детали размыты
- Skip connections передают **высокочастотные детали** с энкодера
- Объединяют глобальный контекст (bottleneck) и локальные детали (skip)

Замечание 22. U-Net и ResNet

Skip connections в U-Net — горизонтальные, между уровнями (разрешениями).
Skip connections в ResNet — вертикальные, внутри одного разрешения.
Обе идеи улучшают gradient flow и качество признаков.

23.12.4 Функция потерь для сегментации

Для семантической сегментации используют:

1. Pixel-wise Cross-Entropy:

$$L = -\frac{1}{H \cdot W} \sum_{i=1}^{H} \sum_{j=1}^{W} \sum_{k=1}^{K} y_{ijk} \log \hat{y}_{ijk}$$

2. Dice Loss (для несбалансированных классов):

$$\text{Dice} = \frac{2|A \cap B|}{|A| + |B|}, \quad L_{\text{Dice}} = 1 - \text{Dice}$$

3. IoU (Intersection over Union):

$$\text{IoU} = \frac{|A \cap B|}{|A \cup B|}$$

Часто используют комбинацию: $L = L_{CE} + L_{Dice}$

23.12.5 Практическая реализация U-Net

```python
import torch
import torch.nn as nn

class DoubleConv(nn.Module):
    """(Conv => BN => ReLU) * 2"""
    def __init__(self, in_channels, out_channels):
        super().__init__()
        self.double_conv = nn.Sequential(
            nn.Conv2d(in_channels, out_channels, 3, padding=1),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace=True),
            nn.Conv2d(out_channels, out_channels, 3, padding=1),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace=True)
        )

    def forward(self, x):
        return self.double_conv(x)

class Down(nn.Module):
    """Downscaling with maxpool then double conv"""
    def __init__(self, in_channels, out_channels):
        super().__init__()
        self.maxpool_conv = nn.Sequential(
            nn.MaxPool2d(2),
            DoubleConv(in_channels, out_channels)
```

```
        )

    def forward(self, x):
        return self.maxpool_conv(x)

class Up(nn.Module):
    """Upscaling then double conv"""
    def __init__(self, in_channels, out_channels):
        super().__init__()
        self.up = nn.ConvTranspose2d(in_channels, in_channels // 2,
                                     kernel_size=2, stride=2)
        self.conv = DoubleConv(in_channels, out_channels)

    def forward(self, x1, x2):
        x1 = self.up(x1)
        # Конкатенация skip connection
        x = torch.cat([x2, x1], dim=1)
        return self.conv(x)
```

Применения U-Net:

- Медицинская сегментация (оригинальная задача)
- Сегментация клеток в микроскопии
- Удаление фона на изображениях
- Сегментация дорог для автопилота
- Сегментация объектов на спутниковых снимках

Ключевые преимущества U-Net:

1. Работает с малым количеством данных (сотни изображений)
2. Skip connections сохраняют детали
3. Симметричная архитектура легко понимается и модифицируется
4. Хорошо обобщается на разные задачи сегментации

23.13. Классы Conv2d и MaxPool2d в PyTorch

23.13.1 torch.nn.Conv2d

Основной класс для сверточных слоев в PyTorch:

```
nn.Conv2d(in_channels, out_channels, kernel_size,
          stride=1, padding=0, dilation=1, groups=1, bias=True)
```

Параметры:

- `in_channels` (int) — число входных каналов (например, 3 для RGB)
- `out_channels` (int) — число выходных каналов (карт признаков)
- `kernel_size` (int или tuple) — размер ядра свертки
- `stride` (int или tuple, по умолчанию 1) — шаг свертки
- `padding` (int или tuple, по умолчанию 0) — дополнение нулями
- `dilation` (int или tuple, по умолчанию 1) — разреженная свертка
- `groups` (int, по умолчанию 1) — число групп для grouped convolution
- `bias` (bool, по умолчанию True) — добавлять ли bias

Формула размера выхода:

$$H_{out} = \left\lfloor \frac{H_{in} + 2 \cdot \text{padding} - \text{dilation} \cdot (\text{kernel\_size} - 1) - 1}{\text{stride}} \right\rfloor + 1$$

Аналогично для W_{out}.

Базовая свертка

```python
import torch
import torch.nn as nn

# Входное изображение: батч=1, каналы=3 (RGB), 32x32
x = torch.randn(1, 3, 32, 32)

# Свертка: 3 входных канала -> 16 выходных, ядро 3x3
conv = nn.Conv2d(in_channels=3, out_channels=16, kernel_size=3)
out = conv(x)

print(out.shape)  # torch.Size([1, 16, 30, 30])
# Размер: (32 - 3 + 1) = 30
```

Свертка с padding и stride

```python
# Свертка с padding=1, stride=2
conv = nn.Conv2d(3, 16, kernel_size=3, stride=2, padding=1)
out = conv(x)

print(out.shape)  # torch.Size([1, 16, 16, 16])
# Размер: floor((32 + 2*1 - 3) / 2) + 1 = 16
```

23.13.2 Параметры padding

Зачем padding?

- Сохранить размер карты признаков
- Избежать потери информации на краях
- Для ядра $k \times k$, padding $p = \lfloor k/2 \rfloor$ сохраняет размер при stride=1

Таблица 21: Влияние padding на размер выхода

Kernel	Padding	Stride	Выход (32×32 вход)
3×3	0	1	30×30
3×3	1	1	32×32 (same)
5×5	0	1	28×28
5×5	2	1	32×32 (same)
3×3	1	2	16×16

23.13.3 Dilation (Разреженная свертка)

Определение 23.26. Dilated Convolution

Увеличивает receptive field без увеличения числа параметров, вставляя "дырки"между элементами ядра.

Эффективный размер ядра:

$$k_{\text{eff}} = k + (k-1) \cdot (d-1)$$

где d — dilation.

Dilated Convolution

```python
# Обычная свертка 3x3: RF = 3
conv_normal = nn.Conv2d(3, 16, kernel_size=3, padding=1)

# Dilated свертка 3x3 с dilation=2: RF = 5
conv_dilated = nn.Conv2d(3, 16, kernel_size=3, padding=2,
 ↪  dilation=2)

out_normal = conv_normal(x)     # torch.Size([1, 16, 32, 32])
out_dilated = conv_dilated(x)   # torch.Size([1, 16, 32, 32])

# Одинаковый размер, но RF dilated больше!
```

23.13.4 torch.nn.MaxPool2d

Pooling слой для уменьшения пространственных размеров:

```python
nn.MaxPool2d(kernel_size, stride=None, padding=0)
```

Параметры:

- `kernel_size` — размер окна pooling
- `stride` — шаг (по умолчанию = kernel_size)
- `padding` — дополнение

Формула размера выхода:

$$H_{out} = \left\lfloor \frac{H_{in} + 2 \cdot \text{padding} - \text{kernel\_size}}{\text{stride}} \right\rfloor + 1$$

MaxPool2d

```
x = torch.randn(1, 16, 32, 32)

# Max Pooling 2x2 с stride=2 (уменьшает в 2 раза)
pool = nn.MaxPool2d(kernel_size=2, stride=2)
out = pool(x)

print(out.shape)  # torch.Size([1, 16, 16, 16])
```

23.13.5 Другие виды Pooling

1. Average Pooling

```
avg_pool = nn.AvgPool2d(kernel_size=2, stride=2)
out = avg_pool(x)  # Усреднение вместо максимума
```

2. Adaptive Average Pooling

```
# Всегда выдает заданный размер выхода
adaptive_pool = nn.AdaptiveAvgPool2d((1, 1))  # Global Average
 ↪  Pooling
out = adaptive_pool(x)  # torch.Size([1, 16, 1, 1])

# Для любого размера входа -> выход 7x7
adaptive_pool = nn.AdaptiveAvgPool2d((7, 7))
out = adaptive_pool(x)  # torch.Size([1, 16, 7, 7])
```

Когда использовать AdaptiveAvgPool?

- Перед полносвязным слоем (вместо flatten)
- Когда размер входа может варьироваться
- Для уменьшения числа параметров в FC

23.13.6 Практический пример: расчет размеров

Цепочка слоев

```
import torch.nn as nn

class SimpleCNN(nn.Module):
    def __init__(self):
        super(SimpleCNN, self).__init__()
        # Вход: 3x32x32
        self.conv1 = nn.Conv2d(3, 16, kernel_size=3, padding=1)  #
         ↪  -> 16x32x32
        self.pool1 = nn.MaxPool2d(2, 2)                          #
         ↪  -> 16x16x16
        self.conv2 = nn.Conv2d(16, 32, kernel_size=3, padding=1) #
         ↪  -> 32x16x16
        self.pool2 = nn.MaxPool2d(2, 2)                          #
         ↪  -> 32x8x8
```

```
        self.fc = nn.Linear(32 * 8 * 8, 10)                    # -> 10

    def forward(self, x):
        x = self.pool1(torch.relu(self.conv1(x)))   # 16x16x16
        x = self.pool2(torch.relu(self.conv2(x)))   # 32x8x8
        x = x.view(x.size(0), -1)                   # flatten: 2048
        x = self.fc(x)                              # 10
        return x

# Проверка размерностей
model = SimpleCNN()
x = torch.randn(1, 3, 32, 32)
out = model(x)
print(out.shape)  # torch.Size([1, 10])
```

23.14. Простая CNN с нуля на CIFAR-10

23.14.1 Датасет CIFAR-10

Определение 23.27. CIFAR-10

Датасет из 60,000 цветных изображений 32×32 в 10 классах:

- 50,000 обучающих изображений
- 10,000 тестовых изображений
- Классы: airplane, automobile, bird, cat, deer, dog, frog, horse, ship, truck

23.14.2 Загрузка и препроцессинг

Загрузка CIFAR-10

```
import torch
import torchvision
import torchvision.transforms as transforms
from torch.utils.data import DataLoader

# Трансформации: нормализация по каналам
transform = transforms.Compose([
    transforms.ToTensor(),  # [0, 255] -> [0, 1]
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))  # [-1, 1]
])

# Загрузка датасета
train_dataset = torchvision.datasets.CIFAR10(
    root='./data', train=True, download=True, transform=transform
)
test_dataset = torchvision.datasets.CIFAR10(
    root='./data', train=False, download=True, transform=transform
)
```

```
19
20 # DataLoader для батчей
21 train_loader = DataLoader(train_dataset, batch_size=64,
    ↪ shuffle=True)
22 test_loader = DataLoader(test_dataset, batch_size=64, shuffle=False)
23
24 print(f"Train: {len(train_dataset)}, Test: {len(test_dataset)}")
25 # Train: 50000, Test: 10000
```

Зачем нормализация?

- Ускоряет сходимость обучения
- Стабилизирует градиенты
- Стандартные значения для CIFAR-10: mean=(0.5, 0.5, 0.5), std=(0.5, 0.5, 0.5)

23.14.3 Архитектура CNN

SimpleCNN для CIFAR-10

```
1  import torch.nn as nn
2  import torch.nn.functional as F
3
4  class CIFAR10_CNN(nn.Module):
5      def __init__(self):
6          super(CIFAR10_CNN, self).__init__()
7          # Сверточные слои
8          self.conv1 = nn.Conv2d(3, 32, kernel_size=3, padding=1)   #
            ↪ 32x32x32
9          self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=1)  #
            ↪ 16x16x64
10         self.conv3 = nn.Conv2d(64, 128, kernel_size=3, padding=1) #
            ↪ 8x8x128
11
12         # Pooling
13         self.pool = nn.MaxPool2d(2, 2)
14
15         # Полносвязные слои
16         self.fc1 = nn.Linear(128 * 4 * 4, 256)
17         self.fc2 = nn.Linear(256, 10)
18
19         # Dropout для регуляризации
20         self.dropout = nn.Dropout(0.5)
21
22     def forward(self, x):
23         # Блок 1: Conv -> ReLU -> Pool
24         x = self.pool(F.relu(self.conv1(x)))  # -> 32x16x16
25
26         # Блок 2: Conv -> ReLU -> Pool
27         x = self.pool(F.relu(self.conv2(x)))  # -> 64x8x8
28
29         # Блок 3: Conv -> ReLU -> Pool
30         x = self.pool(F.relu(self.conv3(x)))  # -> 128x4x4
```

```
        # Flatten
        x = x.view(x.size(0), -1)  # -> 2048

        # FC слои
        x = F.relu(self.fc1(x))
        x = self.dropout(x)
        x = self.fc2(x)

        return x

model = CIFAR10_CNN()
print(f"Параметров: {sum(p.numel() for p in model.parameters()):,}")
```

Параметров: ≈ 270,000

23.14.4 Training Loop

Обучение модели

```
import torch.optim as optim

# Устройство (GPU если доступна)
device = torch.device('cuda' if torch.cuda.is_available() else
 ↪  'cpu')
model = CIFAR10_CNN().to(device)

# Loss и optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

# Training loop
num_epochs = 10
for epoch in range(num_epochs):
    model.train()
    running_loss = 0.0
    correct = 0
    total = 0

    for i, (images, labels) in enumerate(train_loader):
        images, labels = images.to(device), labels.to(device)

        # Forward pass
        outputs = model(images)
        loss = criterion(outputs, labels)

        # Backward pass
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        # Статистика
        running_loss += loss.item()
```

```
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

        if (i + 1) % 200 == 0:
            print(f'Epoch [{epoch+1}/{num_epochs}], '
                  f'Step [{i+1}/{len(train_loader)}], '
                  f'Loss: {running_loss/200:.4f}, '
                  f'Acc: {100*correct/total:.2f}%')
            running_loss = 0.0
```

23.14.5 Тестирование модели

```
model.eval()  # Режим evaluation (отключает dropout)
correct = 0
total = 0

with torch.no_grad():  # Не вычисляем градиенты
    for images, labels in test_loader:
        images, labels = images.to(device), labels.to(device)
        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print(f'Test Accuracy: {100 * correct / total:.2f}%')
```

23.14.6 Улучшения архитектуры

1. **Batch Normalization** после каждой свертки:

```
self.bn1 = nn.BatchNorm2d(32)
x = self.pool(F.relu(self.bn1(self.conv1(x))))
```

2. **Data Augmentation**:

```
transform_train = transforms.Compose([
    transforms.RandomHorizontalFlip(),
    transforms.RandomCrop(32, padding=4),
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])
```

3. **Learning Rate Scheduler**:

```
scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=5,
 ↪  gamma=0.5)
# В конце каждой эпохи:
scheduler.step()
```

4. **Больше эпох и early stopping**