Оптимизация в ML: GD, SGD и метод Ньютона

Содержание

1 Понятие явления переобучения. Способы борьбы с переобучением. Кросс-валидация, её виды.

Переобучение - ситуация, когда модель подстраивается под конкретную выборку, на которой она была обучена, и работает значительно хуже на любых других данных.

Как увидеть переобучение?

- Посмотреть на разделяющую поверхность или на линию регрессии **(1)**. На графиках переобученные модели, так как поверхности прямо по конкретным точкам:

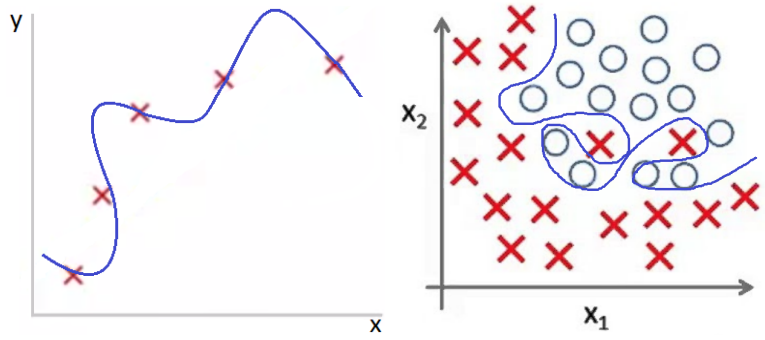


Рис. 1: Линия регрессии(слева) и разделяющая поверхность(справа)

- Посмотреть на разницу между метриками качества на обучении и контроле. Формально степень переобучения можно определить как разность $Metric_{train} - Metric_{test}$. Графически это можно представить на кривых обучения **(2)**:

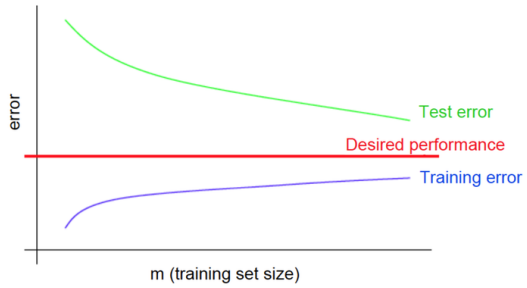


Рис. 2: Кривая обучения

Bias-Variance дилемма и переобучение:

Знаем, что можно разложить качество предсказания модели на составляющие (Bias-Variance decomposition). Ниже пример для метрики MSE:

$$E_{x^n}\left[E_{x,y}[(y-\mu(x^n)(x))^2]\right] = E_{x,y}[(y-E(y|x))^2]\ \textbf{(noise)}+$$
$$E_{x,y}[(E_(y|x)-E_{x^n}\mu(x^n)(x))^2]\ \textbf{(bias)}+$$
$$E_{x,y}\left[E_x^n\left[(\mu(x^n)(x)-E_{x^n}\mu(x^n)(x))^2\right]\right]\ \textbf{(variance)},$$

$\mu(x^n)(x)$ - это ответ модели, *обученной* на выборке из n элементов и *применённой* к наблюдению x. $E[]$ - матожидание выражения, $E[y|x]$ - матожидание y при условии x.

Из выражения видно, что bias - это отклонение нашей усреднённой модели от лучшей из моделей, а variance - это разброс между моделями по всем возможным выборкам и усреднённой модели.

Высокий variance отвечает за переобучение, т.к. отражает зависимость качества модели от выборки, по которой мы её построили.

Из bias-variance decomposition следуют некоторые методы борьбы с переобучением: bagging, boosting.

Как бороться с переобучением:

1. Увеличить количество данных, на которых мы подбираем параметры для модели:
   - Где-то найти ещё выборку
   - Bootstrap - *случайно с возвращением выбираем элементы из нашей выборки.* С помощью усреднённых метрик подбираем параметры.
   - Cross-validation - разбиваем выборку на блоки и используем один из них для контроля, а остальные для обучения. Проделываем это столько раз, сколько блоков, считаем метрику каждый раз и усредняем.
2. Снизить сложность модели:
   - Уменьшить количество фичей или (и) параметров
   - Использовать специфические для класса моделей методы. Пример: pruning - стрижка decission tree, когда мы сначала строим дерево, а потом удаляем узлы с листов. Тут подробнее.
3. Ограничить значения параметров в модели - добавить регуляризацию в наш Loss. Пример: LASSO (L1) и RIDGE (L2).
4. Комбинировать модели:
   - Bagging - агрегируем предсказания независимых моделей (должны быть либо разных видов, либо обучены на разных, независимых данных). Идея в том, что модели работают параллельно и из-за агрегирования корректируют друг друга. Пример: random forest - мы делаем много случайных подвыборок через bootstrap, применяем к ним деревья, у которых случайно выбираются фичи для построения. Их предсказания усредняем.
   - Boosting - последовательно обучаем несколько алгоритмов, чтобы каждый следующий исправлял ошибки предыдущего. Пример: градиентный бустинг.

Кросс-валидация - способ валидации модели, при котором мы делим нашу обучающую выборку на несколько частей и используем одну из частей для контроля, а остальные для обучения модели. Пройдясь по всем частям в цикле и вычислив метрику, получим среднюю точность модели, посчитанную на большем количестве данных, чем первоначальное.

Источник всей инфы далее (много графиков для наглядности и примеры кода)

Принципы разбиения выборки (для любого способа валидации):

1. валидация моделирует реальную работу алгоритма. Следствие: при разбиении выборки должны сохраняться пропорции классов. Пример применения: если алгоритм должен работать на новых пользователях, по которым пока нет данных, то в валидационной выборке должны быть такие пользователи.
2. нельзя явно или неявно использовать метки объектов, на которых оцениваем ошибку (качество). Пример применения: мы сначала разбиваем на обучение и валидацию, и только потом делаем target-encoding, потому что иначе в нашей выборке для обучения неявно будет использоваться информация о таргетах контроля.
3. тест должен быть случайным (или специально подготовленным)

Принципы кросс-валидации: 1) сохраняем баланс классов, 2) рандомизируем, когда разбиваем на части.

Виды кросс-валидации:

- K-fold - алгоритм:
  1. Делим выборку на k примерно равных частей

2. Цикл по i=1,...k: используем i-ю часть для вычисления ошибки, объединение остальных для обучения.
3. усредняем полученные k ошибок

- Leave-one-out - K-fold, где k=количеству наблюдений в выборке
- LeaveOneGroupOut - алгоритм:
  1. Делим выборку на несколько групп в данных (по значениям фичей или по каким-то содержательным соображениям)
  2. Цикл по i=1,...количество выделенных групп: используем i-ю группу для вычисления ошибки, а остальные для обучения модели.
  3. Усредняем полученные ошибки (можно усреднять с весами, пропорциональными размеру группы в выборке, если есть необходимость)
- Out-of-time-контроль - кросс-валидация для временных рядов **(3)**

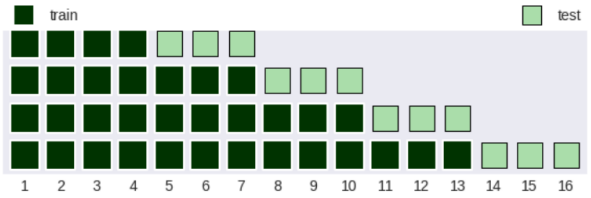


Рис. 3: Кросс-валидация временных рядов

Нюансы K-fold: с увеличением k

- надёжнее оценка качества модели
- обучающая выборка больше походит на остальные данные
- время контроля линейно возрастает
- надо следить за метрикой качества, т.к. не любая адекватно оценивает качество на маленьких подвыборках (хз, какой пример привести)

2 Градиентный спуск (GD), стохастический градиентный спуск (SGD), метод Ньютона

Что показывает градиент. Для дифференцируемой функции потерь $L(w)$ вектор $\nabla L(w)$ указывает направление *наискорейшего роста* функции в точке w, а его норма задаёт локальную скорость роста. Направление *наискорейшего убывания* — это **антиградиент** $-\nabla L(w)$. Так как при обучении мы минимизируем L (функцию потерь), шаги делаются *в сторону антиградиента.*

1) Обычный (batch) градиентный спуск

Правило обновления:

$$w^{(t+1)} \;=\; w^{(t)} \;-\; \eta\,\nabla L\big(w^{(t)}\big).$$

Обозначения: $t = 0, 1, 2, \ldots$ — номер итерации; $w \in \mathbb{R}^d$ — вектор параметров (размерность признакового пространства d); $L(w)$ — функция потерь; $\eta > 0$ — шаг/темп обучения (learning rate). На каждом шаге вычисляется *полный* градиент по всей выборке; стоимость шага порядка $O(Nd)$ (N - число объектов (строк датасета, наблюдений); d - число признаков (столбцов, размерность вектора w)).

2) Стохастический градиентный спуск (SGD, mini-batch)

Полный градиент заменяют несмещённой оценкой по случайному минибатчу $R \subseteq \{1, \dots, N\}$:

$$L(w) = \sum_{i=1}^{N} L_i(w), \qquad L_R(w) = \sum_{i \in R} L_i(w), \qquad w^{(t+1)} \;=\; w^{(t)} - \eta \sum_{i \in R} \nabla L_i\big(w^{(t)}\big).$$

Часто используют средний градиент $\frac{1}{|R|} \sum_{i \in R} \nabla L_i$ (эквивалентно перенастройке η). Цена шага — $O(|R|\, d)$; шаг «шумный», поэтому обычно применяют затухающие/адаптивные схемы для η.

3) Метод Ньютона

Метод учитывает *кривизну* через матрицу Гессе $H(w) = \nabla^2 L(w)$ — матрицу вторых производных:

$$H(L) = \frac{\partial}{\partial w} \left(\frac{\partial L(w)}{\partial w} \right)^{\top} = \begin{bmatrix} \dfrac{\partial^2 L}{\partial w_1^2} & \cdots & \dfrac{\partial^2 L}{\partial w_1 \partial w_d} \\ \vdots & \ddots & \vdots \\ \dfrac{\partial^2 L}{\partial w_d \partial w_1} & \cdots & \dfrac{\partial^2 L}{\partial w_d^2} \end{bmatrix}.$$

Квадратичное разложение Тейлора в точке w даёт

$$L(w + \Delta w) \approx L(w) + g^{\top} \Delta w + \tfrac{1}{2}\, \Delta w^{\top} H\, \Delta w, \qquad g = \nabla L(w).$$

Минимизируя аппроксимацию по Δw, получаем $g + H \Delta w = 0$, откуда

$$\Delta w = -H^{-1} g, \qquad w^{(t+1)} = w^{(t)} - H\big(w^{(t)}\big)^{-1} \nabla L\big(w^{(t)}\big).$$

Замечание. В «чистом» методе Ньютона отдельный learning rate не требуется: матрица H^{-1} сама подстраивает длину и направление шага под кривизну. На практике для устойчивости применяют демпфирование (line search / trust region). Вычислительно хранить и использовать H дорого (память $O(d^2)$, решение $H \Delta w = g$ — до $O(d^3)$), поэтому в больших задачах применяют *квази-Ньютон* (BFGS/L-BFGS) (Это методы, которые не считают гессиан явно, а приближают его (или его обратную) на лету, используя только значения w_k и градиенты $\nabla L(w_k)$).

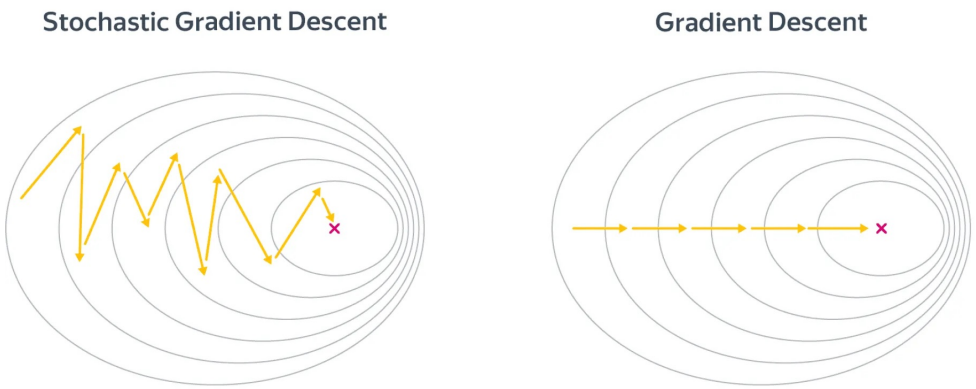


Рис. 4: Сравнение траекторий градиентного спуска (GD, справа) и стохастического градиентного спуска (SGD, слева). У SGD шаги дешевле и более шумные, траектория «дрожит» вокруг направления убывания, но метод быстрее выходит на минимум при больших N. У batch-GD шаги плавные и точные, но каждый обход дороже.

Регуляризация при использовании оптимизаторов

Большинство моделей *уже имеют встроенную регуляризацию* (линейные модели — L1/L2, деревья — ограничения глубины, нейросети — Dropout, weight decay и др.).

Однако при поиске гиперпараметров с помощью оптимизаторов (GridSearch, RandomSearch, Bayesian и т.п.) есть риск **переобучить сам процесс под валидацию**.

Поэтому наряду со встроенной регуляризацией модели нужно применять и **регуляризацию процесса поиска**:

- кросс-валидацию вместо одного train/test split,
- ограничение пространства поиска,
- раннюю остановку в итеративных методах.

Итог: регуляризация в модели защищает сами веса, а регуляризация в оптимизаторе защищает выбор гиперпараметров.

3 Логистическая регрессия: функции для GD, SGD и квази-Ньютона

Модель и обозначения. Пусть есть объекты $x_i \in \mathbb{R}^d$, метки $y_i \in \{0, 1\}$, $i = 1, \ldots, N$. Линейная оценка: $z_i = w^\top x_i + b$. Сигмоида: $\sigma(z) = \dfrac{1}{1 + e^{-z}}$. Вероятность положительного класса: $p_i = \sigma(z_i)$.

Функция потерь. Возьмём отрицательное лог-правдоподобие (бин. кросс-энтропию) с L_2-регуляризацией:

$$\mathcal{L}(w, b) = \sum_{i=1}^{N} \Big[-y_i \log p_i - (1 - y_i) \log(1 - p_i) \Big] \; + \; \frac{\lambda}{2} \, \|w\|_2^2,$$

где $\lambda \geq 0$ — сила регуляризации (на b её обычно не ставят). В `scikit-learn` у `LogisticRegression` параметр `C` связан с λ как *«с точностью до нормировки»* $\lambda \approx 1/C$ (у библиотеки есть внутренняя нормировка по N).

Градиенты. Обозначим $p = (p_1, \dots, p_N)^\top$, $y = (y_1, \dots, y_N)^\top$, $X \in \mathbb{R}^{N \times d}$, строки которого — $x_i^\top$. Тогда

$$\nabla_w \mathcal{L}(w, b) = X^\top (p - y) + \lambda w, \qquad \frac{\partial \mathcal{L}}{\partial b} = \sum_{i=1}^{N} (p_i - y_i).$$

Гессиан. Положим $S = \operatorname{diag}\big(p_i(1 - p_i)\big) \in \mathbb{R}^{N \times N}$. Тогда

$$\nabla^2_{ww} \mathcal{L}(w, b) = X^\top S X + \lambda I_d, \qquad \frac{\partial^2 \mathcal{L}}{\partial w \, \partial b} = X^\top S \mathbf{1}, \qquad \frac{\partial^2 \mathcal{L}}{\partial b^2} = \mathbf{1}^\top S \mathbf{1},$$

где $\mathbf{1}$ — вектор из единиц. На практике удобно «спрятать» свободный член, дополнив матрицу:

$$\tilde{X} = \begin{bmatrix} X & \mathbf{1} \end{bmatrix} \in \mathbb{R}^{N \times (d+1)}, \quad \tilde{w} = \begin{bmatrix} w \\ b \end{bmatrix}, \quad \tilde{\Lambda} = \operatorname{diag}(\lambda, \dots, \lambda, 0).$$

Тогда

$$\nabla_{\tilde{w}} \mathcal{L} = \tilde{X}^\top (p - y) + \tilde{\Lambda} \, \tilde{w}, \qquad \nabla^2_{\tilde{w}\tilde{w}} \mathcal{L} = \tilde{X}^\top S \tilde{X} + \tilde{\Lambda}.$$

GD (batch): формулы обновления

$$w^{(t+1)} = w^{(t)} - \eta \Big[X^\top \big(p^{(t)} - y \big) + \lambda \, w^{(t)} \Big],$$
$$b^{(t+1)} = b^{(t)} - \eta \sum_{i=1}^{N} \big(p_i^{(t)} - y_i \big),$$

где $p^{(t)} = \sigma\big(X w^{(t)} + b^{(t)} \mathbf{1} \big)$. Цена шага: $O(Nd)$.

SGD / mini-batch: формулы обновления

Для случайного батча $R \subset \{1, \dots, N\}$:

$$w^{(t+1)} = w^{(t)} - \eta \Big[X_R^\top \big(p_R^{(t)} - y_R \big) + \lambda \, w^{(t)} \Big],$$
$$b^{(t+1)} = b^{(t)} - \eta \sum_{i \in R} \big(p_i^{(t)} - y_i \big),$$

где X_R — строки X с индексами из R, $p_R^{(t)} = \sigma(X_R w^{(t)} + b^{(t)} \mathbf{1})$. Цена шага: $O(|R| \, d)$.

Ньютон / квази-Ньютон: шаг

В векторной записи с расширением:

$$\Delta \tilde{w} = -\Big(\tilde{X}^\top S \tilde{X} + \tilde{\Lambda} \Big)^{-1} \Big[\tilde{X}^\top (p - y) + \tilde{\Lambda} \, \tilde{w} \Big], \qquad \tilde{w}^{(t+1)} = \tilde{w}^{(t)} + \Delta \tilde{w}.$$

Квази-ньютоновские методы (BFGS/L-BFGS) *не строят* гессиан явно, а итеративно приближают его (или его обратную) по парам $s_k = \tilde{w}_{k+1} - \tilde{w}_k$ и $y_k = \nabla \mathcal{L}_{k+1} - \nabla \mathcal{L}_k$, удовлетворяя секущему условию $B_{k+1} s_k \approx y_k$. В L-BFGS хранятся только последние m пар (s_k, y_k), что даёт память $O(md)$.

Что выбирать на практике. Batch-GD прост, но дорог по N; SGD/mini-batch — дешёвые шумные шаги, требующие планов шага; (L-)BFGS обычно сходится за малое число итераций на гладких выпуклых задачах и удобен через готовые реализации.

4 Алгоритм k Nearest Neighbours

.

Постановка 1 *Пусть есть выборка* $\{x_i\} \in X$*, и множество непересекающихся классов* $\Upsilon = \{1, 2, \ldots, l\}$*, а также метрика* ρ*, и необходимо классифицировать новый объект* $u \in X$

1. Вычислить и расположить элементы в порядке возрастания $\rho(u, x_u^{(1)}) \leq \rho(u, x_u^{(2)}) \leq \cdots \leq \rho(u, x_u^{(\ell)})$
2. Объект относится к тому классу, представителей которого окажется больше всего среди k его ближайших соседей

$$a(u; X^{(\ell)}, k) = \arg_{y \in \Upsilon} \max \sum_{i=1}^{k} \cdot [y_u^{(i)} = y] \tag{1}$$

Гиперпараметры:

- $k \in N$.
- Выбор метрики для расстояния между объектами.
- Если используется ядро, ему тоже надо придумать форму.
- При взвешенной регрессии в уравнение (1) добавляется вес w_i, об этом дальше здесь.

В невзвешенной регрессии предсказания вычисляются согласно формуле $pred(y_i|x_i \in N(x)) =_{a_i} (\sum_{t=1}^{k} I(y(x_t) = a))$.

4.1 Преимущества

- Простой для ручной реализации.

4.2 Недостатки

- Алгоритм действует полным перебором вариантов и не содержит обучения (т.е. изменения самого себя в зависимости от выборки), lazy-learner.
- Куча гиперпараметров, для которых даже сетку не настроить, чтобы их одновременно гипероптимизировать.
- Вычислительная сложность классификации - $O(N \cdot n)$, где N - размер выборки и n - размерность пространства X. Статья Вебера [**?**] показывает, что быстрее чем за линейное время ближайшего соседа просто не удастся отыскать.

4.3 k Centroids

Для каждого класса вычисляются только центры $c_{ij} = \frac{1}{|\{i:y_i=j\}|} \cdot \sum_{i,y_i=j} x_i$ и только они хранятся.

- Не требует экспоненциального количества времени и памяти для хранения, т.к. При этом информация о принадлежности ближайшего окружения точки игнорируется.
- С другой стороны, могут быть классы, чьё множество в исходном пространстве имеет формы типа кольца, и центроиды не несут никакой информации о таких классах.

4.4 Регрессия kNN

.

Так как таргет в kNN-регрессии можно упорядочить, то при построении с помощью алгоритма kNN можно говорить о предпочтении ± плавной зависимости, для которой естественен приоритетный учёт ближайших наблюдений, для kNN-регрессии характерны **весовые схемы** (weighted kNN Generalisations).

Во взвешенной регрессии функционал окажется таким:

$$Q(\alpha, X^l) = \sum_{i=1}^{l} w_{i(x)} \cdot (a - y_i)^2 \to \min_{\alpha \in R}$$

Предсказания вычисляются по формуле Надарая-Ватсона:

$$\hat{y_x} = \frac{\sum_{t=1}^{k} w_t \cdot y(x_t)}{\sum_{t=1}^{k} w_t} \tag{2}$$

Популярными являются весовые схемы:

- $w_t = (k - t + 1)^\delta, \delta > 0$
- $w_t = \frac{1}{t^\delta}$
- Ядра: $w_t = K(\frac{\rho(x,x_t)}{h(x)})$, где $h(x)$ - **возрастающая** по x функция, которая придаёт различие любым различиям в малых расстояниях. Функция K должна давать самые большие значения для самых маленьких аргументов ρ (т.е. ситуации, когда точки совпадают) и околонулевые - для самых маленьких. Например, K может быть просто функцией нормального стандартного распределения:

$$K(x) = \frac{1}{\sqrt{2\pi}} \cdot \exp\left(\frac{-x^2}{2}\right) \tag{3}$$

.

Как находить ближайших соседей? По вычислению расстояния между наблюдениями на основе имеющихся характеристик. Примеры метрик расстояния:

1. $\rho_2(x, y) = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2}$ – Евклидово расстояние;
2. $\rho_1(x, y) = \sum_{i=1}^{n} |x_i - y_i|$ – Манхэттенское расстояние;
3. $\rho_\infty(x, y) = \max\limits_{i \in 1,\ldots,n} |x_i - y_i|$ – Чебышёва расстояние;
4. для отличников: $\rho_p(x, y) = (\sum_{i=1}^{n}(x_i - y_i)^p)^{\frac{1}{p}}$ – расстояние Минковского при $p \geq 1$ является метрикой, при $p < 1$ нарушается нер-во треугольника.

Для категориальных признаков можно поступать как с численными, предварительно применив кодирование: если в признаке есть осмысленный порядок (например, между бакалавром и магистром расстояние ближе, чем между бакалавром и PhDшником), стоит применить ordinal encoder. Если нет определённого порядка, one-hot-encoder выглядит предпочтительнее. Второй метод их анализа описан чуть ниже и не требует применять какое-либо кодирование, просто работа с множествами.

Есть ряд специфичных метрик:

- $p(X, Y) = 1 - \frac{|X \cap Y|}{|X \cup Y|}$ – расстояние Джаккарда (на множествах, удобно использовать в случае наличия только категориальных признаков);
- $p(x, y) = 1 - \frac{\langle x, y \rangle}{||x|| \cdot ||y||}$ – косинусное расстояние (обычно используется для сравнительной оценки схожести какой-либо пары текстов);

- $p(x,y) = \sqrt{(x-y)^T\Sigma^{-1}(x-y)}$ – расстояние Махалонобиса (если рассмотрим диагональную ковариационную матрицу, увидим автоматическую стандартизацию по признакам, которую, к слову, полезно делать при использовании метрических подходов, чтобы учитывать каждый признак в равной степени).

4.5 Проклятие размерности †

Неформальное определение 1 ***Проклятием размерности*** † *(обычно) называется феномен экспоненциального роста необходимого количества памяти и времени на вычисления при наивных алгоритмах классификации, где количество необходимых сравнений расстояний между точками растёт полиномиально/экспоненциально (а не линейно, как при логистической регрессии), что сильно удлиняет их работу.*

5 Постановка задачи кластеризации. Алгоритм K-means.

Задача кластеризации: выявить в данных K кластеров – областей, где объекты внутри одного кластера похожи друг на друга, а объекты из разных кластеров друг на друга не похожи. Похожесть объектов определяется расстоянием между этими объектами, $\rho(x_i, x_j)$, для этого могут быть использованы различные метрики: Евклидова (Минковского - общий вариант), Манхэттена, косинусная (для текстов), Хэмминга и другие.

Построить алгоритм $a : X \to \{1, ...K\}$, определяющий для каждого объекта номер его кластера

K-means

$$Q = \sum_{k=1}^{K} \sum_{i:x_i \in A_k} ||x_i - c_k||^2 \to \min_{c_1,...c_K}$$

Хотим подобрать $c_1, ...c_K$ - центры кластеров, такие что расстояния до $x_i \in A_k$, где $A_k = \{x_i : \rho(x_i, c_k) \leqslant \rho(x_i, c_j) \forall j \neq k\}$ было как можно меньше.

Алгоритм такой (Решение задачи происходит в повторении шагов 1 и 2):

0. Выбор произвольных центров и множеств кластеров
1. Оптимизация функционала по A_k
2. Выбор новых центров c_k, соответсвующих кластерам A_k

Повторяя эти шаги до **сходимости** (пока центры станут неподвижными = нулевое изменение внутриклассового расстояния), мы получим некоторое распределение объектов по кластерам. Новый объект относится к тому кластеру, чей центр является ближайшим.

6 Решающие деревья. Критерии расщепления, выбор оптимальных порогов. Способы борьбы с переобучением в решающих деревьях. Обработка категориальных признаков, обработка пропусков.

Рассмотрим бинарное дерево, в котором:

- каждой внутренней вершине v приписана функция (предикат) $\beta_v : \mathbb{X} \to \{0, 1\}$, обычно $\beta_v(x; j, t) = [x_j < t]$ – бинарное правило;
- каждой листовой вершине v приписан прогноз $c_v \in Y$ (в случае с классификацией листу также может быть приписан вектор вероятностей).

Рассмотрим теперь алгоритм $a(x)$, который стартует из корневой вершины v_0 и вычисляет значение функции β_{v_0}. Если оно равно нулю, то алгоритм переходит в левую дочернюю вершину, иначе в правую, вычисляет значение предиката в новой вершине и делает переход или влево, или вправо. Процесс продолжается, пока не будет достигнута листовая вершина; алгоритм возвращает тот класс, который приписан этой вершине. Такой алгоритм называется **бинарным решающим деревом**.

Критерий расщепления:

$$Q(R, \theta) = H(R) - \frac{|R_{left}|}{|R|} \cdot H(R_{left}) - \frac{|R_{right}|}{|R|} \cdot H(R_{right}), \text{ где меры неоднородности}$$

- для классификации:
  1. $H(R) = \sum_{k=1}^{K} p_k \cdot (1 - p_k)$ – критерий Джини
  2. $H(R) = -\sum_{k=1}^{K} p_k \cdot \log p_k$ – энтропийный критерий
  3. $H(R) = 1 - \max\limits_{1,\dots,K} p_k$ – miss-classification criteria
- для регрессии:
  1. $H(R) = \frac{1}{|R|} \sum_{(x_i,y_i)\in R} \left(y_i - \frac{1}{|R|} \sum\limits_{(x_j,y_j)\in R} y_j \right)^2$ – MSE
  2. $H(R) = \frac{1}{|R|} \sum_{(x_i,y_i)\in R} \left| y_i - \underset{(x_j,y_j)\in R}{Median}\ y_j \right|$ – MAE

Напоминание: параметр θ обозначает рассматриваемое правило для ветвления.

6.1 Методы борьбы с переобучением:

1. Критерии останова
   Эти критерии позволяют не достраивать дерево до самого конца, когда в каждом листе останется ровно по одному элементу.
   - Ограничение максимальной глубины дерева
   - Ограничение минимального числа объектов в листе
   - Ограничение максимального количества листьев в дереве
   - Останов в случае, если все объекты в листе относятся к одному классу
   - Требование, что функционал качества при дроблении улучшался как минимум на s процентов
2. Стрижка дерева
   Стрижка дерева является альтернативой критериям останова, описанным выше. При использовании стрижки сначала строится переобученное дерево (например, до тех пор, пока в каждом листе не окажется по одному объекту), а затем производится оптимизация его структуры с целью улучшения обобщающей способности.

Одним из методов стрижки является cost-complexity pruning. Обозначим дерево, полученное в результате работы жадного алгоритма, через T_0. Поскольку в каждом из листьев находятся объекты только одного класса, значение функционала $R(T)$ будет минимально на самом дереве T_0 (среди всех поддеревьев). Однако данный функционал характеризует лишь качество дерева на обучающей выборке, и чрезмерная подгонка под нее может привести к переобучению. Чтобы преодолеть эту проблему, введем новый функционал $R_\alpha(T)$, представляющий собой сумму исходного функционала $R(T)$ и штрафа за размер дерева:
$R_\alpha(T) = R(T) + \alpha|T|$,
где $|T|$ — число листьев в поддереве T, а $\alpha > 0$ — параметр. Это один из примеров регуляризованных критериев качества, которые ищут баланс между качеством классификации обучающей выборки и сложностью построенной модели. Можно показать, что существует последовательность вложенных деревьев с одинаковыми корнями:
$T_K \subset T_{K-1} \subset ... \subset T_0$,
здесь T_K — тривиальное дерево, состоящее из корня дерева T_0, в которой каждое дерево T_i минимизирует критерий $R_\alpha(T)$ для α из интервала $\alpha \in [\alpha_i, \alpha_i + 1)$, причем $0 = \alpha_0 < \alpha_1 < ... < \alpha_K < \infty$.
Эту последовательность можно достаточно эффективно найти путем обхода дерева. Далее из нее выбирается оптимальное дерево по отложенной выборке или с помощью кросс-валидации.

3. Использование в ансамблях (например, случайный лес)

6.2 Обработка категориальных признаков

Категориальный признак $x_j \in categories \equiv \{cat_1, \ldots, cat_k\}$. Его значение для конкретного наблюдения из выборки (x_j^i) означает определенную категорию, к которой наблюдение причислено. Мы хотим заменить каждое значение категориального признака некоторым числом, иными словами – закодировать, чтобы далее строить дерево с использованием привычных сплитов по числовым фичам.

- **Target Encoding**
  Суть: Значения категориальной переменной кодируется средним таргетом по категориям $encoding(cat_j) = \frac{\sum_i [x_j^i = cat_m] \cdot y_i}{\sum_i [x_j^i = cat_m]}$. Фактически, $encoding : category \in categories \mapsto value \in \mathbb{R}$.
  Проблема: При такой кодировке возникает target leakage (информация о таргете передается в фичи). Мы строим дерево на фичах, которые содержат в себе информацию о таргете, что ведет к переобучению.
  Решение проблемы: Подбираем числовое значение для кодировки определенной категории с помощью кросс-валидации. Разбивем на фолды и для каждого фолда считаем число для кодировки значения категории по описанному выше принципу, но уже не по всему датасету, а по остальным фолдам. И так делается для каждого фолда. Можно делать иначе и просто добавлять шум, но этот способ может быть хуже.
- **Label Encoding**
  Суть: Значения категориальной переменной кодируется порядковым номером. В лекции было сказано, что принцип особо не важен, например: в таком порядке в каком категории встречаются в датасете, такой номер и присваивается конкретной категории.
  Проблема: При такой кодировке возникает порядок там, где мы его не ожидаем, что

может влиять на предсказательную силу обучаемого дерева.
Комментарий: Кажется, сюда подойдет то, что написано в файле Соколова про обработку категориальных переменных. Можно упорядочить категории в фиче x_j по среднему значению таргета внутри категории, то есть $cat_{(1)}, \dots, cat_{(k)}$: $\frac{\sum_i [x_j^i = cat_{(1)}] \cdot y_i}{\sum_i [x_j^i = cat_{(1)}]} \leq \dots \leq \frac{\sum_i [x_j^i = cat_{(k)}] \cdot y_i}{\sum_i [x_j^i = cat_{(k)}]}$. Дальше кодируем эти категории по принципу $cat_{(i)} \mapsto i$ и можем проводить разбиения вершин как с обычными числовыми признаками. В файле Соколова указано, что так можно делать и внутри конкретной вершины дерева.

- **One Hot Encoding**
Суть: Фактичеки, это кодирование дамми переменными. То есть если для признака x_j есть всего k значений категорий, то есть $x_j \in categories \equiv \{cat_1, \dots, cat_k\}$, то создается k новых признаков, каждый из которых является индикатором ($x_{ohe_m} \equiv [x_j = cat_m], m \in \{1, \dots, k\}$).
Проблема: Данные могут сильно разростаться. Кроме этого, в каждой новой фиче, которая по сути вляется индикатором определенной категории, обычно достаточно мало единиц и больше нолей, что влияет на качество построения дерева (а конкретно в момент подсчета функционала качества при разбиении вершин).

6.3 Обработка пропущенных значений

- *Игнорируем пропуски*
Пусть нам нужно вычислить функционал качества для предиката $\beta(x) = [x_j < t]$, но в выборке R для некоторых объектов не известно значение признака j — обозначим их через V_j.
В таком случае при вычислении функционала можно просто проигнорировать эти объекты, сделав поправку на потерю информации от этого:
$Q(R, j, s) \approx \frac{|R \backslash V_j|}{|R|} Q(R \backslash V_j, j, s)$.
Затем, если данный предикат окажется лучшим, поместим объекты из V_j как в левое, так и в правое поддерево. Также можно присвоить им при этом веса $\frac{|R_l|}{|R|}$ в левом поддереве и $\frac{|R_r|}{|R|}$ в правом. В дальнейшем веса можно учитывать, добавляя их как коэффициенты перед индикаторами $[y_i = k]$ во всех формулах.
- *Обработка пропусков при прогнозировании*
Допустим, нам надо спрогнозировать таргет для наблюдения, у которого значение одной из фичей пропущено. Если объект попал в вершину, предикат которой не может быть вычислен из-за пропуска, то прогнозы для него вычисляются в обоих поддеревьях, и затем усредняются с весами, пропорциональными числу обучающих объектов в этих поддеревьях. Иными словами, если прогноз вероятности для класса k в поддереве R_m обозначается через $a_{mk}(x)$, то получаем такую формулу:

$$a_{mk}(x) = \begin{cases} a_{\ell k}(x), & \beta_m(x) = 0 \\ a_{rk}(x), & \beta_m(x) = 1 \\ \frac{|R_\ell|}{|R_m|} a_{\ell k}(x) + \frac{|R_r|}{|R_m|} a_{rk}(x), & \beta_m(x) \text{ нельзя вычислить.} \end{cases}$$

Либо можно просеить наблюдение по другому признаку. То есть у нас есть обученное дерево, просеивая через него объект мы сталкиваемся с проблемой в одной из вершин, что дальше мы просеить его не можем, так как значение признака, который используется в предикате в этой вершине, пропущено для нашего наблюдения. Тогда для этой вершины можно построить предикат с использованием другого признака

(значение которого не пропущено для наблюдения, таргет которого надо предсказать), который даст похожее разбиение наблюдений из обучающей выборки в этой вершине. Такой предикат можно использовать и он называется "суррогатный предикат".

- *Простые методы*
  Иногда гораздо более простые методы показывают не менее качественные результаты: замена пропущенных значений на ноль; на число, которое больше максимального по фиче.

7 Определение отступа

(Из Соколова) Величина $M_i = y_i\langle w, x_i\rangle$, называемая отступом (margin). Знак отступа говорит о корректности ответа классификатора (положительный отступ соответствует правильному ответу, отрицательный — неправильному), а его абсолютная величина характеризует степень уверенности классификатора в своём ответе. Напомним, что скалярное произведение $\langle w, x_i\rangle$пропорционально расстоянию от разделяющей гиперплоскости до объекта; соответственно, чем ближе отступ к нулю, тем ближе объект к границе классов, тем ниже уверенность в его принадлежности.

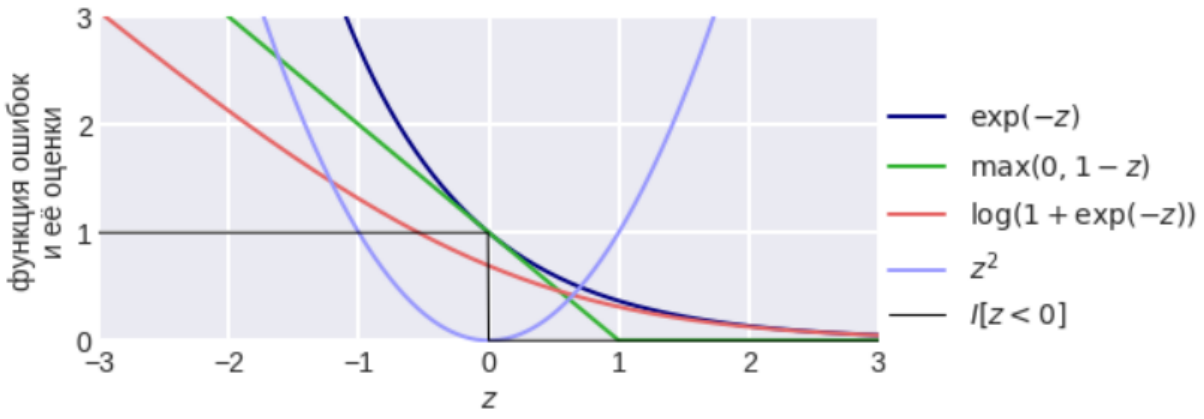
То есть, суть отступа в том, чтобы не просто говорить модели, что она ошиблась, а чтобы давать степень ее ошибки для обновления весов. Изначально отступ - это идея в дискретном случае, которая выдает просто $0/1$ в случае правильно/непраивльного ответа. Соотвественно, есть функции для оптимизации и более точной оценки ошибки (см. ниже).

Также стоит отметить, что в SVM классификаторе веса подбираются таким образом, что отступ по умолчанию равен 1 для удобства и интерпретируемости. То есть, если отступ объект (после пропуска через функцию) лежит от 0 до 1, то он находится внутри отступа, если больше 1, то он классифицирован неправильно, что сильно штрафуется. В SVM регрессоре параметр отступа выбирается от 0 до 1 и говорит, какие значения отступа мы считаем незначительными и не учитываем как ошибку.

7.1 Различные виды верхних оценок на функционал доля ошибок. Логлосс, Hinge-loss

Полученный выше функционал $[M_i < 0]$ невозможно оптимизировать с помощью, например, градиентного спуска, так как он дискретный. Вместо него работают с верхними оценками - гладкими функциями, среди которых выделяют:

- Log-loss - $log(1 + e^{-M})$ (выводится из логистической регрессии, см. 6 билет)
- Hinge-loss - $max(0, 1 - M)$ (выводится из Soft-margin постановки SVM, см. 8 билет)



8 Постановка задачи SVM (Hard и Soft margin).

Hard-margin SVM:

$$\begin{cases} \frac{1}{2}\|w\|^2 \to \min\limits_{w,b} \\ y_i(\langle w, x\rangle + b) \geq 1 \end{cases}$$

$a(x) = sign(\langle w, x\rangle + b)$
Если $\lambda_i > 0$ и $y_i(\langle w, x_i\rangle + b) = 1$, то такой объект обучающей выборки x_i называется *опорным вектором*. Решение задачи зависит только от опорных векторов.

Soft-margin SVM:

$$\begin{cases} \frac{1}{2}\|w\|^2 + C\sum \xi_i \to \min\limits_{w,b,\xi_i} \\ y_i(\langle w, x_i\rangle + b) \geq 1 - \xi_i \\ \xi_i \geq 0 \end{cases}$$

Здесь объекты выборки можно разделить на 3 категории:

1. Эталонные
   $\lambda = 0, \xi = 0$, Не влияют на веса и на решение.
2. Опорные $\xi = 0, 0 < \lambda < c$
3. Нарушители $\xi > 0, \lambda = c$. Причем нарушители бывают 2 видов:
   (a) $\xi < 1$ Правильно классифицированы, но маленькая margin.
   (b) $\xi > 1$ Неправильно классифицированы.

Решение задачи зависит от опорных векторов и векторов-нарушителей.

Чем больше значение коэффициента регуляризации C, тем на меньшее количество объектов будет опираться гиперплоскость, чтобы не занижать функционал, но тогда переобучение.

Откуда берется $\frac{1}{2}\|w\|^2 \to \min\limits_{w,b}$?
Хотим максимизировать расстояние от x_i до разделяющей гиперплоскости, то есть $\frac{|w^T x + b|}{\|w\|} \to \max\limits_{w,b}$. Веса в этом выражении подбираются так, чтобы числитель=1, тогда это сводится к задаче $\|w\|^2 \to \min$ и отсюда же возникает ограничение ≥ 1. Сама найденная гиперплоскость никак не зависит от этой нормировки числителя.

9 DBSCAN

DBSCAN - еще один алгоритм кластеризации (умеет работать с шумными данными).
Согласно алгоритму выборка разбивается на три типа точек (красные, желтые, синие) в зависимости от параметров ϵ и $min\_samples$ - гиперпараметры. Основная идея алгоритма заключается в том, что для каждой точки **кластера** окрестность радиуса (ϵ) должна содержать как минимум $min\_samples$ точек.
алгоритм Точки делятся следующим образом:

- точка помечается как «основная» , если в ее ϵ окрестности есть $min\_samples$ точек
- точка помечается как «достижимая», если в ее ϵ окрестности нет $min\_samples$ точек, но есть точки, помеченные как «основные»
- точка помечается «выбросом», если для нее не выполняются оба условия выше

Кластер образуют «основные» и «достижимые» точки (последние будут лежать на границе кластера). Если назначить ϵ слишком большим алгоритм построит 1 большой

кластер. Если слишком маленьким и при этом $min\_samples = 0$ -> все точки-отдельные кластеры. Если ϵ мал , а $min\_samples > 0$ -> алгоритм определит почти все точки , как выбросы.
Преимущества алгоритма : умеет распознавать кластеры различной формы (не только сферической, как в основном делает алгоритм k-means), хорошо работает на данных, в которых есть шум.

10 Критерии качества кластеризации

Похожесть объектов определяется метриками качества. Метрики качества (или критерии качества) показывают на сколько правильно работает алгоритм. Существует 2 вида критериев : **внутренние** и **внешние**.
Внешние требуют использование дополнительных данных (например, информацию об истинных данных). Если известно истинное распределение объектов по классам, то можно рассматривать задачу кластеризации, как задачу многоклассовой классификации, и в качестве внешних критериев использовать F-меру, например.
Внутренние метрики основаны на свойствах выборки и кластеров. Пусть c_k - центр k-ого кластера. Примеры внутренних метрик:

- 1. Внутрикластерное расстояние - суммарное расстояние между объектами внутри кластера по всем кластерам (это расстояние нужно минимизировать).

$$\sum_{k=1}^{K}\sum_{i=1}^{n}[a(x_i) = k]\rho(x_i, c_k)$$

- Межкластерное расстояние - сумма расстояний между каждой парой объектов из разных кластеров (это расстояние нужно максимизировать - объекты из разных кластеров должны быть менее похожи).

$$\sum_{i,j=1}^{n}[a(x_i) \neq a(x_j)]\rho(x_i, x_j)$$

- Индекс Данна (формула с лекции Антона) - отношение внутрикластерного и межкластерного расстояния (хотим минимизировать)

$$\text{Dunn Index} = \frac{\rho_{\text{внутрикластерное}}}{\rho_{\text{межкластерное}}}$$

- Силуэт - показывает, насколько среднее расстояние до объектов своего кластера отличается от среднего расстояния до объектов других кластеров.Силуэтом выборки называется средняя величина силуэта объектов данной выборки. Введем обозначения :
  a — среднее расстояние от данного объекта до объектов из того же кластера, b — среднее расстояние от данного объекта до объектов из ближайшего кластера (отличного от того, в котором лежит сам объект).
  Силуэт для отдельного объекта:

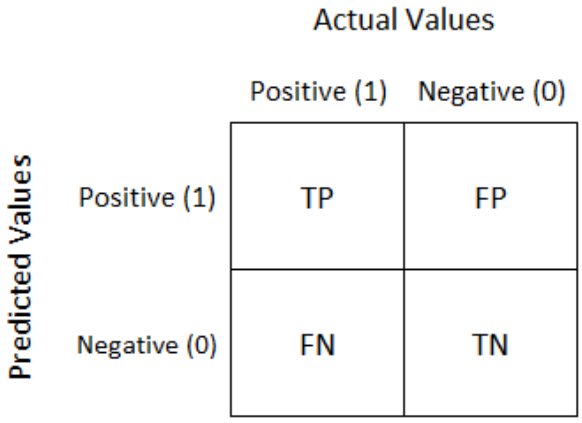
$$Silhouette_i = \frac{b - a}{max(a, b)}$$

  Расчитываем силуэт для каждого объекта выборки и усредняем. Данная величина лежит в диапазоне [-1,1]. Значения, близкие к -1, соответствуют плохим (разрозненным) кластеризациям, значения, близкие к нулю, говорят о том, что кластеры пересекаются и накладываются друг на друга, значения, близкие к 1, соответствуют «плотным» четко выделенным кластерам.

11 Метрики качества классификации и регрессии. Матрица ошибок. Метрики в случае многоклассовой задачи классификации. Рок-кривая, площадь под ней, явная формула для рок-кривой.

11.1 Матрица ошибок

Матрица ошибок – матрица, размера KxK, где K – количество классов, которая показывает точность классификации модели. В случае двух классов:



Многоклассовый случай. Пусть есть выборка X размера n, y_i – метки класса для каждого объекта i (всего K-штук классов). Матрицей ошибок называется следующая матрица:

$$M = m_{jk}, \text{ размерности KxK}$$

$$m_{jk} = \sum_{i=0}^{N} [a(x_i) = j][y_i = k], \text{ где j – класс, предсказанный классификатором.}$$

11.2 Метрики качества для задач регрессии

- MSE $= \frac{1}{n}\sum_{i=1}^{n}(y_i - a_i)^2$ (где взять: `sklearn.metrics.mean_squared_error`)
- RMSE $= \sqrt{\text{MSE}}$ (где взять: `sklearn.metrics.mean_squared_error(..., squared=False)`; в новых версиях также `sklearn.metrics.root_mean_squared_error`)
- MAE $= \frac{1}{n}\sum_{i=1}^{n} w_i\,|y_i - a_i|$ (где взять: `sklearn.metrics.mean_absolute_error` с параметром `sample_weight`)
- $-\log(L) = -\log($функции правдоподобия$)$ (где взять: `SciPy` — `scipy.stats` и соответствующий `logpdf` распределения, суммировать по наблюдениям; в `scikit-learn` общей метрики NLL для регрессии нет)
- MAPE $= \frac{1}{n}\sum_{i=1}^{n}\frac{|y_i - a_i|}{|y_i|}$ (где взять: `sklearn.metrics.mean_absolute_percentage_error`)

11.3 Метрики качества для задач классификации

- **Функция правдоподобия (бернуллиевская)**:

$$L = \prod_{i=1}^{n} a_i^{y_i}\,(1 - a_i)^{1-y_i}, \qquad -\log L = -\sum_{i=1}^{n}\big[y_i \log a_i + (1 - y_i)\log(1 - a_i)\big].$$

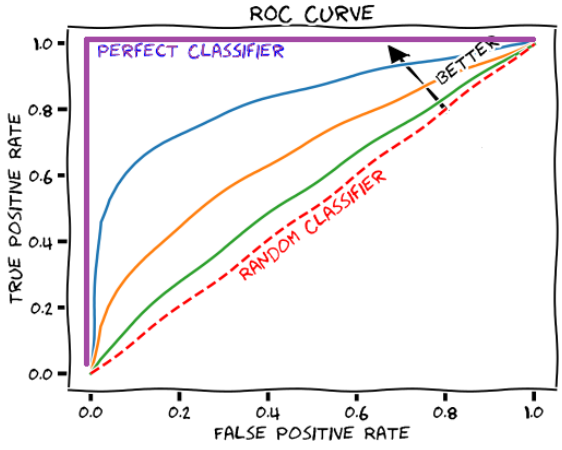
Показывает, насколько согласованы предсказанные вероятности $a_i = P(\hat{y}_i = 1 \mid x_i)$ с метками $y_i \in \{0, 1\}$; максимизация L (или минимизация $-\log L$) эквивалентна минимизации бинарной кросс-энтропии. (где взять: `sklearn.metrics.log_loss`)

- **Accuracy** $= \dfrac{TP + TN}{TP + TN + FP + FN}$. Доля правильно классифицированных объектов среди всех; уместна при сбалансированных классах. (где взять: `sklearn.metrics.accuracy_scor`
- **Precision** $= \dfrac{TP}{TP + FP}$. Доля истинно положительных среди всех предсказанных положительных; «насколько чисты» позитивные предсказания. (где взять: `sklearn.metrics.precision_score` с `average` для многокласса)
- **Recall** $= \dfrac{TP}{TP + FN}$. Доля найденных положительных среди всех реально положительных; «насколько полно» обнаружены позитивы. (где взять: `sklearn.metrics.recall_score` с `average` для многокласса)
- **F1** $= \dfrac{2}{\frac{1}{\text{precision}} + \frac{1}{\text{recall}}} = \dfrac{2\,\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$. Гармоническое среднее precision и recall; удобен при дисбалансе классов. (где взять: `sklearn.metrics.f1_score` с `average` для многокласса)
- **ROC-AUC**. Площадь под ROC-кривой (TPR против FPR при изменении порога); равна вероятности, что случайный позитив получает больший скор, чем случайный негатив. (где взять: `sklearn.metrics.roc_auc_score`; для самой кривой — `sklearn.metrics.roc_curve`)

Где: TP — верно предсказанные положительные, TN — верно предсказанные отрицательные, FP — ложноположительные, FN — ложноотрицательные.

11.4 ROC кривая

ROC кривая – кривая зависимости *False Positive Rate* (FPR) $= \dfrac{FP}{FP + TN}$ (сколько из всех отрицательных не были найдены) и *True Positive Rate* (TPR) $= \dfrac{TP}{TP + FN}$ (сколько из всех положитльных оветов были найдены). AUC ROC – площадь под этой кривой, значение характеризует долю правильно классифицируемых пар. Для ее построения ранжируем вероятность присвоения класса ($b(x)$), для каждого порога считаем FPR и TPR.



$$AUC = \frac{\sum_i \sum_j [y_i < y_j] \times [a(x_i) < a(x_j)]}{\sum_i \sum_j [y_i < y_j]} = \frac{1}{n_- \times n_+} \times \sum_{i<j} [y_i < y_j]$$

ROC может пойти по диагонали (не ступенькой), если у одинаковых наблюдений будут разные метки. Самый плохой случай AUC: AUC = 0.5, т.к. по сути ничего не упорядочили и угадываем в 50% случаев. Если AUC ROC = 0, то значит, что присвоенные метки полностью перепутаны и достаточно поменять их местами.

Критерий AUC-ROC имеет большое число интерпретаций: например, он равен вероятности того, что случайно выбранный положительный объект окажется позже случайно выбранного отрицательного объекта в ранжированном списке, порожденном $b(x)$ (AUC (площадь под кривой) = вероятность того, что модель присвоит случайному положительному объекту больший "скор уверенности в положительности", чем случайному отрицательному).

Кривая ROC не всегда ступенчатая: точки могут соединяться и наклонными линиями.

11.5 Метрики в случае многоклассовой задачи классификации

В многоклассовых задачах, как правило, стараются свести подсчет качества к вычислению одной из рассмотренных выше двухклассовых метрик. Выделяют два подхода к такому сведению: микро- и макро-усреднение.

Пусть выборка состоит из K классов. Рассмотрим K двухклассовых задач, каждая из которых заключается в отделении своего класса от остальных, то есть целевые значения для k-й задачи вычисляются как $y_i^k = [y_i = k]$. Для каждой из них можно вычислить различные характеристики алгоритма $a_i^k(x) = [a(x) = k]$.

При микро-усреднении сначала эти характеристики усредняются по всем классам, а затем вычисляется итоговая двухклассовая метрика - например, точность, полнота или F-мера. Например, точность будет вычисляться по формуле

precision(a,X) = $\frac{\overline{TP}}{\overline{TP}+\overline{FP}}$

где, например, $\overline{TP}$ вычисляется по формуле $\overline{TP} = \frac{1}{K}\sum_{k=1}^{K} TP_k$

При макро-усреднении сначала вычисляется итоговая метрика для каждого класса, а затем результаты усредняются по всем классам. Например, точность будет вычислена как

precision(a,X) = $\frac{1}{K}\sum_{k=1}^{K} precision_k(a, X)$

где precision_k(a, X) = $\frac{TP_k}{TP_k+FP_k}$

В макро-усреднении каждый класс вносит равный вклад вне зависимости от размера класса.

12 Регрессии

12.1 Линейная регрессия

Алгоритм выглядит следующим образом:

$$a(x) = \langle w, x_i \rangle + b$$

где $w, x \in \mathbb{R}^d$, $b = const$. Если предполагается наличие константного признака в X, то можно переписать $a(x) = \langle w, x_i \rangle$ (или $a(x) = w^T x$). Алгоритм настраивается путем минимизации суммы квадратов отклонений (Родной метод наименьших квадратов):

$$RSS = \sum_{i=1}^{n} e_i^2 = \sum_{i=1}^{n} (y_i - a(x_i))^2 = \sum_{i=1}^{n} (y_i - \langle w, x_i \rangle)^2 \to \min_w$$

В результате оптимизации получаем веса: $w = (X^T X)^{-1} X^T y$ Частный случай для регрессии с одним x и константой: $w = \dfrac{cov(x, y)}{var(x)}$ При множественной регрессии может возникать проблема вырожденности матрицы $w = (X^T X)^{-1} X^T y$, которая может быть решена следующими способами:

- Регуляризация - например, с l2-регуляризацией $w = (X^T X + \lambda I)^{-1} X^T y$
- Отбор признаков - умный перебор подмножества признаков, оценка качества признаков (фильтры), встроенные методы (ex: LASSO)
- Уменьшение размерности (например, PCA)
- Увеличение выборки

12.2 Определение логистической регрессии

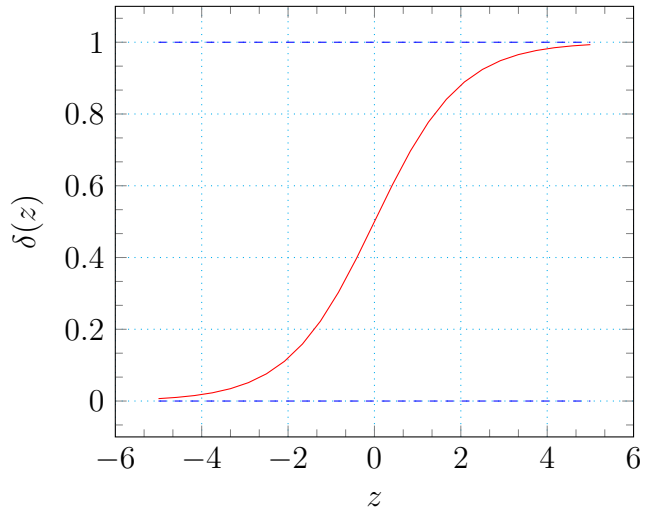
Логистическая регрессия - это метод обучения, который получается в результате использования логистической функции потерь.

Мы решаем задачу принадлежности объекта к классу $+1$: $a(x) = p(y = +1|x)$

Зададим классификатор следующим образом: $a(x) = \delta(\langle w, x \rangle) = \frac{1}{1+\exp(-\langle w,x \rangle)}$, где соответственно функция сигмоиды выглядит следующим образом: $\delta(z) = \frac{1}{1+\exp(-z)}$. Линии уровня сигмоиды - гиперплоскости, поэтому данный классификатор является линейным.

График сигмоиды выглядит следующим образом:

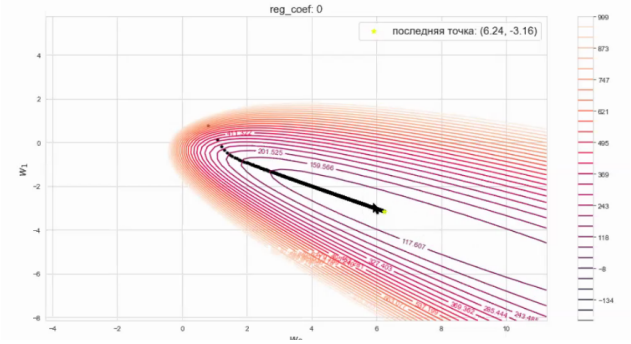
График сигмоиды



Ответы классификатора $a(x) \in (0, 1)$ - их можно интерпретировать как вероятность принадлежности к классу $+1$.

12.3 Регуляризация функционала в логистической регрессии

При решении, например, методом градиентного спуска, можно столкнуться с проблемой наличия плато. Плато - область, в которой функция примерно постоянна и везде достигает своего минимума. Его стоит изберать - иначе во-первых, метод градиентного спуска будет долго сходиться и, во-вторых, так значения функционала примерно одинаковые и шаги градиентного спуска слишком маленькие, метод никуда не выйдет. Плато выглядят так:



Как бороться с плато? Используем регуляризацию:

$$\tilde{Q} = \sum_{i=1}^{n} \log(1 + \exp(-y_i \langle w, x_i \rangle)) + \underbrace{\frac{\lambda}{2} \parallel w \parallel_2^2}_{l_2\text{-регуляризация}} \to \min_w$$

Для чего нужна регуляризация?

- Регуляризация борется с плато - делает линии уровня функционала более кругообразными
- Допустим, найдена гиперплоскость, которая идеально разделяет множества. Будем умножать ее веса w на увеличивающиеся положительные величины $w \to \infty$ - гиперплоскость не изменится. Тогда $-y_i \langle w, x \rangle \to \infty$. При этом функционал (без регуляризации) будет уменьшаться $\tilde{Q} = \sum\limits_{i=1}^{n} \log(1 + \exp(-y_i \langle w, x_i \rangle)) \to 0$. С точки зрения нашей задачи ничего не изменится (гиперплоскость). Если добавить регуляризацию, то при $w \to \infty$, тогда $\frac{\lambda}{2} \parallel w \parallel^2 \to \infty$ - поэтому у нас не будут бесконечно увеличиваться веса

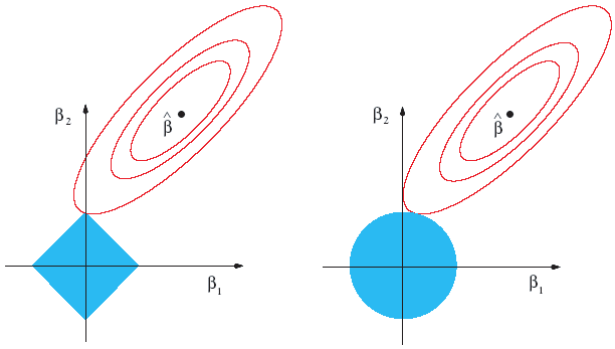
Кроме того, есть l_1-регуляризация.

$$\tilde{Q} = \sum_{i=1}^{n} \log(1 + \exp(-y_i \langle w, x_i \rangle)) + \underbrace{\frac{\lambda}{2} \parallel w \parallel_1}_{l_1\text{-регуляризация}} \to \min_w$$

- Для l_1-регуляризации мы используем Манхеттенскую норму: $\parallel w \parallel_1 = \sum\limits_{j=1}^{d} |w_j|$. В таком случае точка решения будет лежать в одном из углов ромба (см. картинку ниже - слева). Она позволяет отбирать признаки - какая-та часть коэффициентов будет равна 0 (это случится если не слишком маленькая лямбда, в противном таком случае - ромб будет большой, залезет на овал и там будет оптимум, то есть признаки не занулятся)
- Для l_2-регуляризации мы используем Евклидово расстояние. Точка решения будет лежать в любом месте на окружности (см. картинку ниже - справа).

- Еще один вид регуляризации: elsatic net (нечасто используется). Линии уровня выглядят как юла:

$$\tilde{Q} = \sum_{i=1}^{n} \log(1 + \exp(-y_i \langle w, x_i \rangle)) + \underbrace{\gamma_1 \parallel w \parallel_1 + \gamma_2 \parallel w \parallel_2^2}_{elasticnet} \to \min_w$$



Для l_2-регуляризации найдем градиент от регуляризатора $\tilde{Q}$:

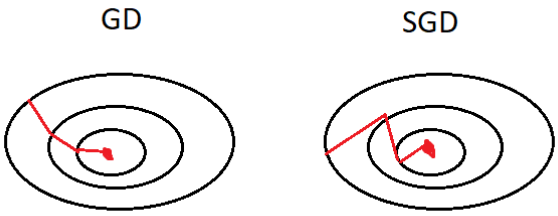
$$\nabla_w \tilde{Q} = \frac{1}{n} \sum_{i=1}^{n} -y_i x_i \delta(-y_i \langle w, x \rangle) + \lambda w$$

Аналогично для стохастического градиентного спуска:

$$\tilde{\nabla}_w \tilde{Q} = \frac{1}{|N|} \sum_{i \in N} -y_i x_i \delta(-y_i \langle w, x \rangle) + \lambda w$$

где N - элементы батча. SGD может блуждать больше, чем GD (см. картинку ниже), но в его направление аналогичное. Скорость схождения у SGD примерно такая же, как и у GD, иногда слегка больше. Если объектов слишком много, то GD нереализуем (не хватит оперативной памяти), т таких случаях помогает SGD.

Различия в поисках решений наглядно выглядят так:



13 Ядра в краткой практике: что это, где применять и как выбирать

Идея без формул. Ядро — это функция похожести между объектами, которая соответствует скалярному произведению в некотором (скрытом) пространстве признаков. Трюк в том, что мы не строим эти новые признаки явно: алгоритм (например, SVM/SVR) использует только значения ядра $K(x_i, x_j)$. За счёт этого можно моделировать нелинейные зависимости так же просто, как линейные — но в другом пространстве.

Где встречается.

- Классификация и регрессия: `SVC`, `SVR`, Kernel Ridge.
- Обучение без учителя: Kernel PCA, спектральная кластеризация.
- Байесовские модели: гауссовские процессы (ядро = ковариационная функция).

Как думать о разных ядрах. Каждое ядро — это способ сказать, когда два объекта «похожи».

- **Linear** (`kernel="linear"`): похожесть по направлению признаков. Работает, когда зависимость почти линейная или признаки высокоразмерные и разреженные (тексты, мешок слов).
- **RBF/гауссово** (`kernel="rbf"`): локальная похожесть; две точки похожи, если близки в евклидовой метрике. Универсальная «база по умолчанию».
- **Polynomial** (`kernel="poly"`): учитывает взаимодействия признаков до заданной степени. Полезно, когда ожидаются невысокие полиномиальные эффекты.
- **Sigmoid** (`kernel="sigmoid"`): исторически связан с однослойной нейросетью (tanh). Используется редко, чувствителен к настройкам.
- Другие: лапласовское (как RBF, но по L1), Matérn (часто в гауссовских процессах), строковые ядра (для ДНК/текста), предвычисленное (`"precomputed"`).

Как выбирать на практике.

- **Высокая размерность, разреженность (тексты, n_features $\gg$ n_samples):** берите `linear`. Он быстрый и хорошо калибруется.
- **Нелинейные границы при умеренных данных (до десятков тысяч объектов):** начните с `rbf`. Это надёжный универсальный выбор.
- **Известны конкретные взаимодействия малой степени:** попробуйте `poly` со степенью `degree=2` или `3`.
- **Гауссовские процессы и пространственные данные:** ядра Matérn, RBF, периодические.
- **Доменные объекты (строки, графы):** специализированные ядра или `precomputed` с собственной метрикой.

Главные гиперпараметры.

- **Для RBF**: `gamma` контролирует «радиус влияния» точки. Малое γ — гладкая граница (может недообучать), большое γ — очень извилистая граница (риск переобучения). Хорошие стартовые сетки: $\gamma \in \{10^{-3}, 10^{-2}, 10^{-1}, 1, 10\}$ с обязательным масштабированием признаков.
- **Для polynomial**: `degree` (степень полинома), `coef0` (сдвиг), `gamma` (масштаб). Увеличение `degree` быстро повышает сложность границы.
- **Общий для SVM/SVR**: `C` регулирует баланс между шириной зазора и ошибками. Большой `C` стремится «объяснить» каждую точку (риск переобучения), малый `C` — даёт более широкий зазор (лучшее обобщение).
- **Для SVR**: ещё `epsilon` — ширина «трубки» без штрафа; увеличивает устойчивость к шуму.

Диагностика и типичные ошибки.

- **Масштабирование**: для RBF/poly обязательно используйте `StandardScaler` или аналог, иначе `gamma` будет «не про то».
- **Перебор сетки**: подбирайте `C`, `gamma` (и `degree`/`coef0` для poly) по лог-сетке с кросс-валидацией.

- **Признаки в разных шкалах**: приводите к единому масштабу, иначе ядро переоценит «большие» признаки.
- **Слишком большая** γ: почти идеальная подгонка на трейне, резкое падение на валидации. Снижайте γ и/или `C`.
- **Стоимость по памяти/времени**: ядровые методы хранят матрицу $n \times n$ и масштабируются примерно квадратично/кубически по числу объектов. Для $n \gtrsim 50\,000$ используйте линейные ядра или аппроксимации (Nyström, Random Fourier Features).

Аппроксимации и предвычисленные ядра.

- **Быстрые приближения RBF**: `RBFSampler` (случайные фурье-признаки), `Nystroem`. Позволяют обучать линейную модель на расширенных признаках и получать эффект RBF быстрее.
- **Собственное ядро**: `SVC(kernel="precomputed")` ожидает квадратную симметричную матрицу похожести на трейне и прямоугольную — на тесте. Старайтесь, чтобы ядро было положительно полуопределённым (иначе могут возникать нестабильности).

14 PCA

Первым делом для использования алгоритма нужно отцентрировать данные (то, о чём говорилось в предыдущей части – вычесть среднее значение). В пакетах обычно зашивают стандартизацию, то есть при этом каждое наблюдение ещё и делится на стандартное отклонение признака. Ещё на лекции говорили, что есть альтернатива: загнать наблюдения в отрезок от 0 до 1 (вычесть минимум по выборке и поделить на разницу между максимума и минимума). Какой из этих методов лучше не всегда понятно, но всегда можно проверить на практике.

Классическая стандартизация: $x_i = \frac{x_i - \overline{x}}{\sigma_x}$

Альтернатива: $x_i = \frac{x_i - x_{min}}{x_{max} - x_{min}}$

Мы хотим найти минимальное среднее расстояние расстояние от каждого x_i до плоскости L:

$$Q = \frac{1}{n}\sum_{i=1}^{n} \rho(x_i; L) \to \min_{L}$$

То есть расстояние до проекции:

$$\frac{1}{n}\sum_{i=1}^{n} \|x_i - AA^T x_i\|^2 \to \min_{A=a_1,\cdots,a_d}$$

Такая оптимизационная задача нам дает два преимущества:

1. Мы можем выбрать расположение подпространства
2. Мы можем выбрать число векторов d, по которым будет происходить минимизация

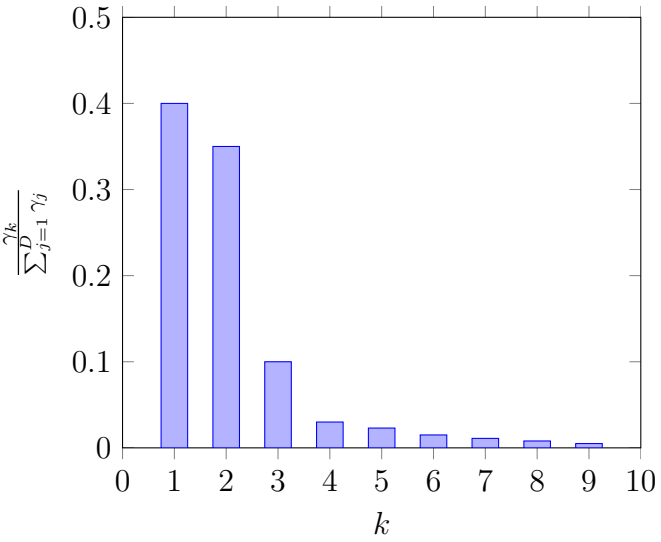
14.1 Критерии отбора оптимального числа главных компонент

Сортируем все собственные значения от большего к меньшему.

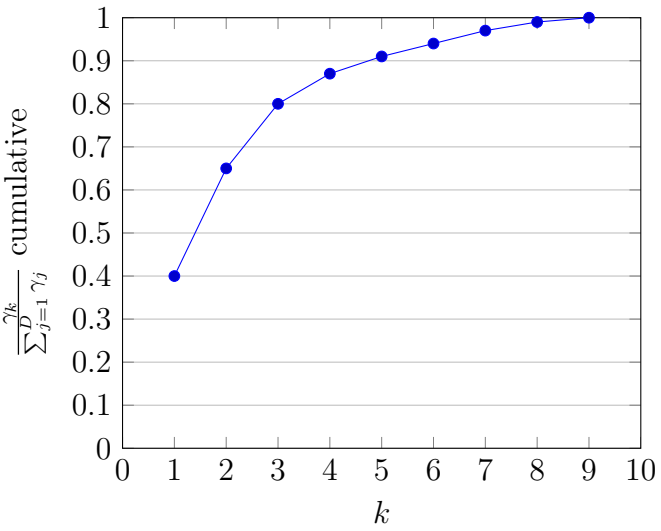
1. Строим картинку, где по x — номер собственного значения, по y — величина собственного значения (можно поделённую на сумму всех значений). Смотрим на нее. Там, где наблюдается резкое уменьшение абсолютной величины собственного значения, то дальше мы значения мы не берем (на картинке 1 последнее значение d=2).

2. Можно нарисовать картинку, где по оси x будут также номера собственных значений, а по y — их накопленная доля от суммы всех собственных значений. (картинка 2). Когда эта доля станет нас устраивать, например, 3 признака объясняют 80 процентов дисперсии, то тогда отбираем соответствующее количество признаков.
3. Если нужно визуализировать данные, то 2 или 3 признака.
4. Если есть исходное знание о данных. Например, есть биржевой индекс, который зависит в основном от курса акций трех компаний, тогда и новая размерность будет 3.
5. Собственно, если мы используем PCA в какой-либо задаче/модели, то число главных компонент можно рассматривать как обычный гиперпараметр, и искать такое значение, при котором используемая нами метрика качества принимает максимальное значение.

Картинка 1



Картинка 2



15 Построение ансамблей. Бэггинг и случайный лес, Градиентный бустинг, Stacking.

15.1 Построение ансамблей

Ансамблем (Ensemble, Multiple Classifier System) называется алгоритм, который состоит из нескольких алгоритмов машинного обучения, а процесс построения ансамбля называется ансамблированием (ensemble learning).

Простейший пример ансамбля в регрессии – усреднение нескольких алгоритмов:

$$a(x) = 1/n * (b_1(x) + b_2(x) + ... + b_k(x))$$

Для понимания: Для построения ансамбля достаточно в каком-то количестве взять какие-то алгоритмы и как-то соединить их ответы: случайный лес - это ансамбль деревьев. Усреднённые Knn также будет ансамблем. Бустинг - это тоже типичный пример ансамлбля. Нейронная сеть - это тоже ансамбль (однородный).

15.2 Бэггинг и случайный лес

Бэггинг (**B**ootstrap **Agg**regat**ing**) – способ построения ансамбля следующим образом: мы строем базовую модель на бутстреп подвыборке, делаем это несколько на разных подвыборках, получаем разные модели, после чего применяем к данным и усредняем ответы

При отборе подвыборки с помощью бутстрепа вероятность какого-либо элемента из изначального множества попасть в подвыборку составляет при каком-либо взятии $\frac{1}{n}$, тогда вероятность не попасть выборку при этом взятии $1 - \frac{1}{n}$. Отсюда получаем, что вероятность не попасть в бутстреп подвыборку того же размера, что и изначальная составляет $\left(1 - \frac{1}{n}\right)^n$, значит, вероятность попасть хотя бы раз будет:

$$1 - \left(1 - \frac{1}{n}\right)^n \longrightarrow_{n\to\infty} 1 - \frac{1}{e} \approx 0,63$$

Получается, что каждая подвыборка состоит примерно из 63% оригинальной выборки, на которой и обучается, остальная часть называется out-of-bag-наблюдениями. На этих наблюдениях можно оценивать качество моделей, считая для них ответы, а с их помощью и ошибки. Ответ для таких наблюдений получаем следующим образом:

$$a_{OOB}(x_j) = \frac{1}{|\{i : x_j \in OOB_i\}|} \sum_{i:x_j \in OOB_i} b_i(x_j)$$

Случайный лес
Алгоритм:

1) Берём бутстрап подвыборку

2) Строим на подвыборке случайное дерево следующим образом: сначала выбираем случайное подмножество фичей, потом по ним ищем оптимальный сплит, после чего опять выбираем случайное подмножество фичей и уже по ним делаем оптимальный сплит и так далее

3) Повторяем шаги 1), 2) для постройки необходимого нам количества деревьев
4) Получаем ответы случайного леса, как среднее предсказание по всем деревьям

15.3 Градиентный бустинг

Бустинг — метод ансамблирования, в котором мы хотим получить алгоритм следующего вида:

$$a_N(x) = \sum_{j=0}^{N} b_j(x)$$

$b_j(x)$ - это простые алгоритмы (weak learners), которые обучаются итеративно (то есть сначала мы назначаем нулевой алгоритм $b_0(x)$, а потом последовательно обучаем первый, второй и т.д.) по следующему правилу:

$$\left[\begin{array}{rcl} s_i^{(N)} & = & y_i - a_N(x_i) \\ b_N(x) & = & \underset{b \in \mathcal{A}}{\operatorname{argmin}} \sum_{i=1}^{n} (b(x_i) - s_i^{(N)})^2 \end{array}\right.$$

Теперь для градиентного бустинга немного изменим модель добавив в неё веса γ_j:

$$a_N(x) = \sum_{j=0}^{N} \gamma_j b_j(x)$$

Применим нашу loss-функцию на наш алгоритм и воспользуемся свойством того, что мы составные алгоритмы обучаем по очереди:

$$\sum_{i=1}^{n} \mathcal{L} y_i, a_N(x_i)) \to \min$$

$$\sum_{i=1}^{n} \mathcal{L} y_i, a_{N-1}(x_i) + \gamma_N b_N(x_i)) \to \min_{\gamma_N b_N}$$

Если loss-функция квадратичная ($\mathcal{L}(y,z) = \frac{1}{2}(y-z)^2$), то тогда получим, что $s_i^{(N)}$ является антиградиентом этой loss-функции в точки предсказания нашей модели:

$$s_i^{(N)} = -\frac{\partial}{\partial z} \mathcal{L}(y,z) \Big|_{z=a_{N-1}(x_i)}$$

Сделаем замену $\gamma_N b_N(x_i) = s_i$. Тогда получим, что максимизация будет устроена следующим образом:

$$\sum_{i=1}^{n} \mathcal{L} y_i, a_{N-1}(x_i) + s_i) \to \min_{s_1,\ldots,s_n}$$

$$s_i = y_i - a_{N-1}(x_i)$$

Однако такой подход будет плох, так как он не учитывает метрику (работает хорошо с квадратичной), и будет возникать переобучение. Тогда наилучшим выбором будет следовать по антиградиенту loss-функции.

Тогда применим формулу Тейлора для случая многомерной функции и поставим оптимизационную задачу:

$$f(x+s) = f(x) + Df(x)^T s + o(||s||)$$

$$\begin{cases} f(x+s) \to \min\limits_s \\ ||s|| = 1 \end{cases}$$

Из минимизации получаем:

$$s = -\frac{Df(x)}{||Df(x)||}$$

Алгоритм градиентного бустинга:

1) Вычисляем антиградиенты нашей функции потерь в точках ответов нашей предыдущей компазиции (которая уже настроена)

$$s_i^{(N)} = -\frac{\partial}{\partial z}\mathcal{L}(y,z)\Big|_{z=a_{N-1}(x_i)}$$

2) Настраиваем наши слабые модели (weak learners)

$$b_N(x) = \underset{b\in\mathcal{A}}{\operatorname{argmin}} \sum_{i=1}^{n} (b(x_i) - s_i^{(N)})^2$$

3) Настраиваем веса (гамму)

$$\gamma_N(x) = \underset{\gamma\in R}{\operatorname{argmin}} \sum_{i=1}^{n} \mathcal{L}(y_i, a_{N-1}(x_i) + \gamma b_N(x_i))$$

Особенности градиентного бустинга над деревьями:

Когда мы используем деревья, то мы можем представить простую модель (дерево) в следующем виде:

$$b(x) = \sum_{t=1}^{T} w_t[x \in R_t]$$

Тогда наша функция потерь выглядит следующим образом:

$$\sum_{i=1}^{n} \mathcal{L}(y_i, a_{N-1}(x) + \gamma_N \sum_{t=1}^{T} w_{Nt}[x \in R_t])$$

Однако и γ_N и w_{Nt} представляют из себя константы, а потому нет смысла сначала искать одну, потом другую, проще их сразу объединить, после чего мы можем разбить нашу задачу на эквивалентные подзадачи, так как константа ищется для каждого листа отдельно, независимо от других:

$$\sum_{i:(x_i,y_i)\in R_t} \mathcal{L}(y_i, a_{N-1}(x_i) + w_{Nt})$$

Тогда алгоритм получается следующий:

1) Вычисляем антиградиенты нашей функции потерь в точках ответов нашей предыдущей компазиции (тут ничего не поменялось, так как антиградиенты зависят от функции потерь, а не от моделей)

$$s_i^{(N)} = -\frac{\partial}{\partial z}\mathcal{L}(y, z)\Big|_{z=a_{N-1}(x_i)}$$

2) Настраиваем наши деревья (которые соответсвуют weak learners, и которые обучаются по антиградиентам)

$$b_N(x) = \underset{b\in DecisionTree}{\operatorname{argmin}} \sum_{i=1}^{n} (b(x_i) - s_i^{(N)})^2$$

3) Обновляем константы в листьях в соответствии с нашей функцией потерь (то есть деревья у нас обучаются и выдают константу оптимальную для MSE, а мы их меняем а не оптимальные с точки зрения функции потерь)

$$w_{Nt}^*(x) = \underset{w\in R}{\operatorname{argmin}} \sum_{i:(x_i,y_i)\in R_t} \mathcal{L}(y_i, a_{N-1}(x_i) + w)$$

Пример для понимания: из второго шага мы получили константы, и так как у нас во втором шаге минимизируется квадрат разницы, то константа равна среднему значению, теперь представим, что в качестве функции потерь у нас взят модуль отклонения, тогда получим, что оптимальная константа для функции потерь должна быть равна медианному значению

15.4 Stacking

Стекинг (Stacked Generalization или Stacking) — один из самых популярных способов ансамблирования алгоритмов, т.е. использования нескольких алгоритмов для решения одной задачи машинного обучения. Основная идея стекинга - это построение метапризнаков, то есть ответов базовых алгоритмов на объектах выборки, и обучение на них мета- алгоритма. В общем виде формально записывается в следующем виде:

Это функция от ответов других алгоритмов

$$a(x) = d(b_1(x), b_2(x), ..., b_k(x))$$

Алгоритм принимает на вход ответы других алгоритмов, как признаки

Графическая иллюстрация стэкинга

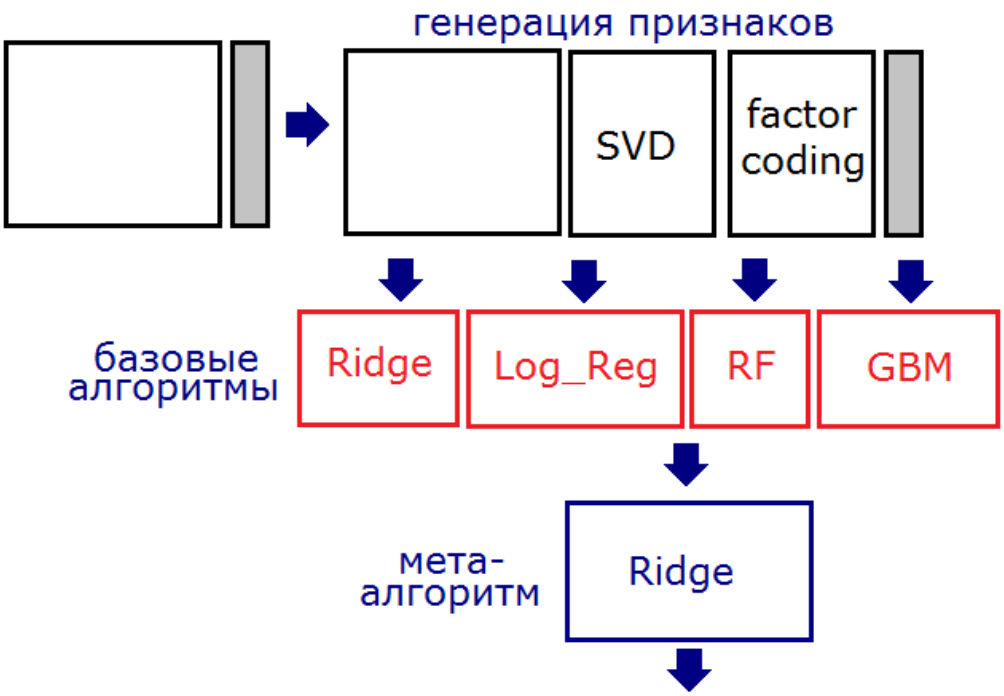


Рис. 5: Стандартное изображение стекинга

Простейшая схема стекинга — блендинг (Blending): обучающую выборку делят на две части. На первой обучают базовые алгоритмы. Затем получают их ответы на второй части и на тестовой выборке. Понятно, что ответ каждого алгоритма можно рассматривать как новый признак (т.н. «метапризнак»). На метапризнаках второй части обучения настраивают метаалгоритм. Затем запускают его на метапризнаках теста и получают ответ. В рамках блендинга ответы алгоритмов просто усредняются, тогда как в стекинге они используются как входные данные для нового алгоритма.

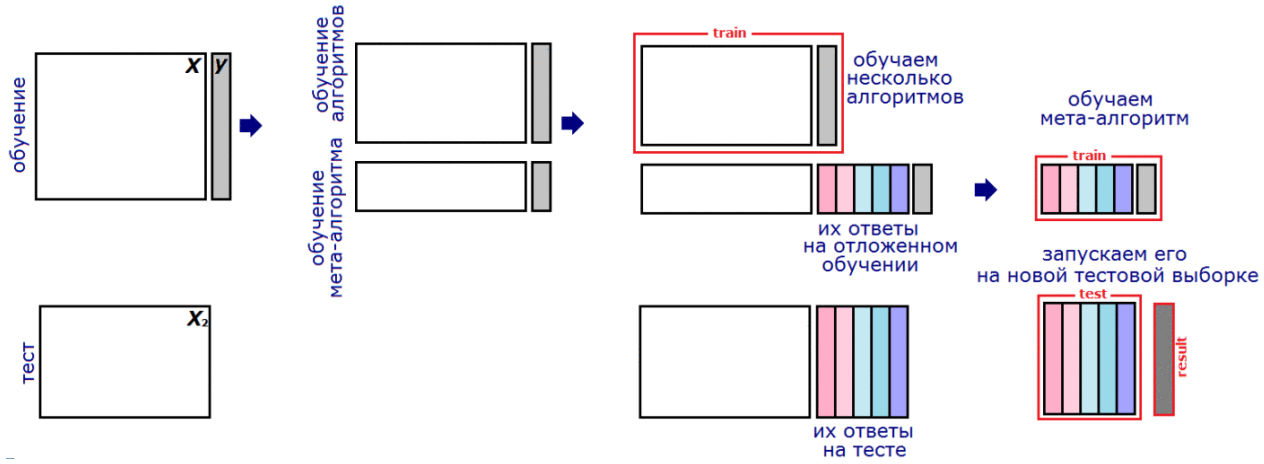


Рис. 6: Схема классического блендинга

Заметна проблема нерационального использования выборки, поэтому для повышения качества надо усреднить несколько блендингов с разными разбиениями обучения. Вместо усреднения иногда конкатенируют обучающие (и тестовые) таблицы для метаалгоритма, полученные при разных разбиениях: здесь мы получаем несколько ответов для каждого объекта тестовой выборки — их усредняют. На практике такая схема блендинга сложнее в реализации и более медленная, а по качеству может не превосходить обычного усреднения.

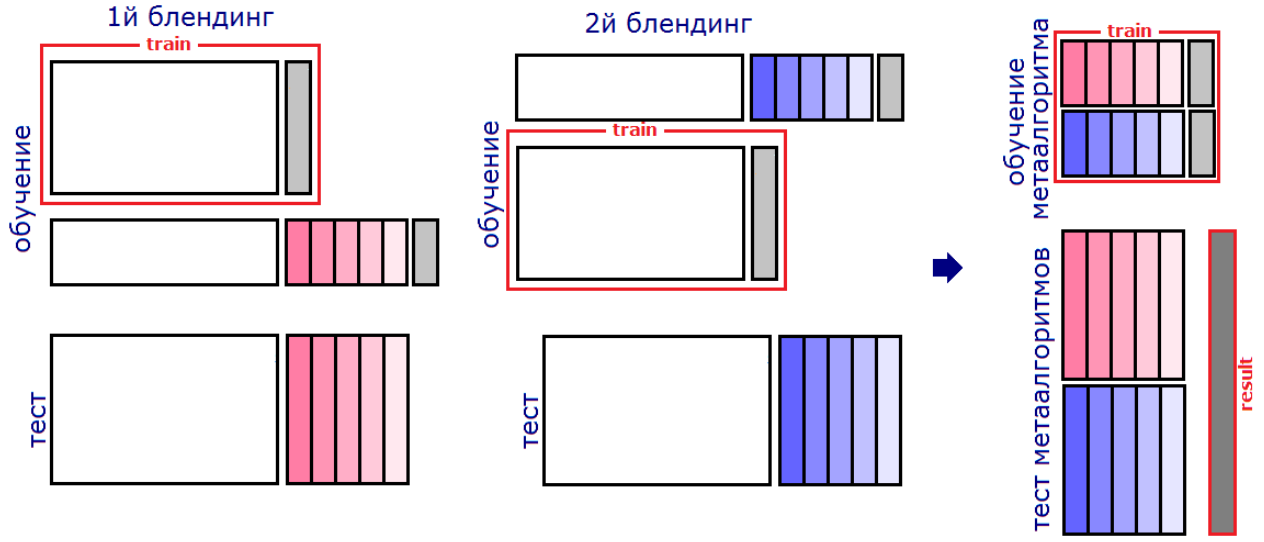


Рис. 7: Возможная модификация блендинга

В рамках классического стэкинга выборку разбивают на части (фолды), затем последовательно перебирая фолды обучают базовые алгоритмы на всех фолдах, кроме одного, а на оставшемся получают ответы базовых алгоритмов и трактуют их как значения соответствующих признаков на этом фолде. Для получения метапризнаков объектов тестовой выборки базовые алгоритмы обучают на всей обучающей выборке и берут их ответы на тестовой. Самый главный недостаток (классического) стекинга в том, что метапризнаки на обучении (пусть и полноценном — не урезанном) и на тесте разные.Часто с указанным недостатком борются обычной регуляризацией.

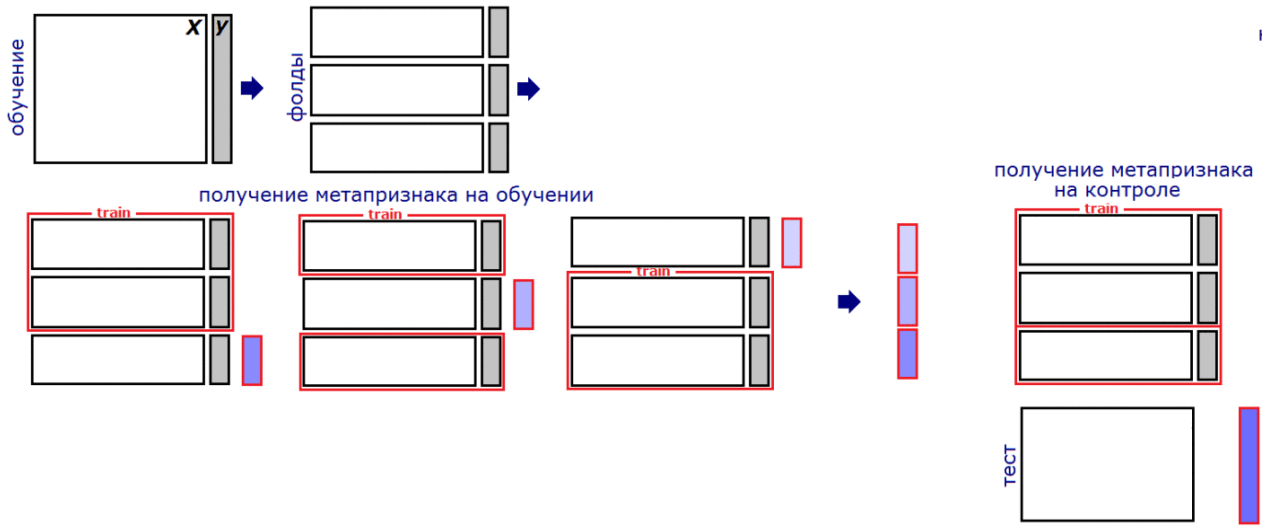


Рис. 8: Получение метапризнака в классическом стекинге

16 CatBoost vs XGBoost vs LightGBM: краткая сравнилка (без кода)

Деревья и способ роста.

- **CatBoost:** строит симметричные (oblivious) деревья фиксированной глубины: на каждом уровне один общий сплит. Даёт стабильность, компактные модели и быстрый инференс; иногда уступает по пиковому качеству на «сложных» данных.
- **XGBoost:** классический бустинг по деревьям, обычно *level-wise* (растит уровни равномерно). Хороший баланс скорости и аккуратности, предсказуемое поведение.
- **LightGBM:** *leaf-wise* (best-first) рост с ограничением по глубине: агрессивно углубляет «самый выгодный» лист. Часто выше качество при меньшем времени, но без аккуратной регуляризации риск переобучения выше.

Работа с категориальными признаками.

- **CatBoost:** нативная поддержка категорий (без ручного OHE/TE). Использует ordered target statistics/CTR с перестановками и *ordered boosting*, что снижает утечки и переобучение.
- **XGBoost / LightGBM:** как правило, требуют кодирования (OHE/TE/WOE и т. п.). В LGBM есть встроенная категорика через `categorical_feature` (histogram-based split по категориям), но чувствительна к кардинальности и настройкам; в XGB «нативная» категорика появилась позже и тоже требует аккуратной подготовки.

Алгоритмы сплитов и ускорения.

- **CatBoost:** собственный *ordered boosting* + симметричные деревья; устойчив «из коробки», особенно на категориальных фичах.
- **XGBoost:** histogram- и exact-сплиты, L1/L2-регуляризация, shrinkage (`eta`), `subsample`/`colsampl` эталон стабильности и прозрачности.
- **LightGBM:** GOSS (Gradient-based One-Side Sampling) и EFB (Exclusive Feature Bundling) для ускорения на больших/разреженных матрицах; очень экономен по памяти и времени.

Контроль переобучения и регуляризация.

- **CatBoost:** ordered boosting уменьшает утечки; «разумные» значения по умолчанию — часто хорошо работает без тонкого тюнинга.
- **XGBoost:** богатые регуляры (`reg_lambda`, `reg_alpha`), shrinkage, `subsample`/`colsample_bytree`; легко контролировать сложность.
- **LightGBM:** сильные регуляры + leaf-wise рост ⇒ критичны `min_data_in_leaf`, `num_leaves`, `feature_fraction`, `bagging_fraction`; без них можно «перерасти».

Скорость и ресурсы.
- **CatBoost:** часто быстрее XGB на наборах с большим числом категориальных фич; на чисто числовых — сопоставим, иногда медленнее LGBM.
- **XGBoost:** «золотая середина»: не самый быстрый, но стабильно оптимизирован (CPU/GPU).
- **LightGBM:** чаще всего самый быстрый на больших высокоразмерных числовых данных, экономен по памяти.

Пропуски, дисбаланс, ранжирование.
- **Пропуски:** все три обрабатывают `NaN` (hist-based направляют пропуски в «лучшую» ветвь).
- **Дисбаланс:** у всех есть способы масштабировать классы/веса (`scale_pos_weight`, sample weighting, bagging).
- **Задачи:** ранжирование, мультикласс, регрессия поддерживаются всеми; CatBoost особенно силён в ranking и мультиклассе с категориками.

Интерпретируемость и продакшн.
- **CatBoost:** симметричные деревья ⇒ быстрый инференс; встроенные плоты и удобная работа с категориями; компактные модели.
- **XGBoost:** самый «прозрачный» в тюнинге, много tooling'а, зрелые интеграции.
- **LightGBM:** отличное соотношение скорость/качество на больших числовых данных; требует аккуратности с гиперпараметрами.

Когда что выбрать.
- **Много категориальных признаков / средний объём данных** ⇒ CatBoost (меньше препроцессинга, меньше утечек).
- **Смешанные данные / нужен предсказуемый baseline и прозрачный тюнинг** ⇒ XGBoost.
- **Большие, высокоразмерные числовые данные / важны скорость и память** ⇒ LightGBM.

Чувствительность к гиперпараметрам (в целом).
- **CatBoost:** часто «работает из коробки»; тюнят глубину/итерации/learning rate/регуляризацию; для категорий достаточно передать списки фич.
- **XGBoost:** ключевые — `learning_rate`, `max_depth`, `min_child_weight`, `subsample`, `colsample_bytree`, `reg_lambda`/`reg_alpha`.
- **LightGBM:** критичны `num_leaves`, `max_depth` (или без него), `min_data_in_leaf`, `feature_fraction`, `bagging_fraction`, `learning_rate`; аккуратно с `num_leaves` и `min_data_in_leaf`, чтобы не «перерасти».

17 EM-алгоритм. На какие компоненты раскладывается неполное правдоподобие. Как выглядят шаги EM-алгоритма.

Expectation maximization:
EM-алгоритм – итерационный метод максимизации правдоподобия выборки. Метод оптимизации в вероятностных моделях с исходной задачей: $\log p(x|\theta) \to \max\limits_{\theta}$, где $x \in R^n$, θ – набор параметров. Такой функционал не всегда легко оптимизировать, проще оптимизировать другой функционал, являющейся его нижней оценкой, используя введение скрытой переменной z с плотностью q и функцию полного правдоподобия ($\log p(x, z|\theta)$). $KL(p||q)$ – дивергенция Кульбака-Лейблера, характеризующая расстояние между двумя распределениями.

$$\log p(x|\theta) = \log p(x|\theta) \cdot \int q(z)dz = \int q(z) \cdot \log p(x|\theta)dz = \{p(x,y) = p(x|y) \cdot p(y)\} =$$
$$= \int q(z) \cdot \log \frac{p(x,z|\theta)}{p(z|x,\theta)}dz = \int q(z) \cdot \log \frac{p(x,z|\theta) \cdot q(z)}{q(z) \cdot p(z|x,\theta)}dz =$$
$$= \int q(z) \cdot \log \frac{p(x,z|\theta)}{q(z)}dz + \int q(z) \cdot \log \frac{q(z)}{p(z|x,\theta)}dz =$$
$$= \mathcal{L}(q,\theta) + KL(q(z)||p(z|x,\theta)) \geqslant \mathcal{L}(q,\theta)$$

$\mathcal{L}(q,\theta)$ – это ELBO (evidence lower bound), его и будем оптимизировать.

Так, можем оптимизировать нижнюю границу в 2 шага по q и θ

Шаг E: $\mathcal{L} \to \max\limits_{q} \sim KL(p||q) \to \min\limits_{q}$, т.к. $\mathcal{L} = \log p(x|\theta) - KL(q(z)||p(z|x,\theta))$, минимум достигается при совпадении распределений q и p: $q^* = p(z|x,\theta^{old})$

Шаг M: $\mathcal{L}(q^*,\theta) \to \max\limits_{\theta}, \mathcal{L}(q^*,\theta) = \int q^*(z) \times \log \frac{p(x,z|\theta)}{q^*(z)}dz = E_{q^*(z)} \log p(x,z|\theta) + H(q^*)$: $\theta^{new} = \underset{\theta}{\text{argmax}} E_{q^*(z)} \log p(x,z|\theta)$

18 Bias-variance-decomposition формула разложения. Интерпретация компонент.

Пусть есть какое-то распределение фичей (x) и соответствующих им ответов (y): $p(x,y)$ – *плотность распределения*. Из этого распределения генерируется выборка X и ответы на ней. Будем рассматривать задачу регрессии (y – вещественное число). Пусть a – наш алгоритм. В качестве функции потерь будем рассматривать квадротичную функцию : $L(y,a) = (y-a(x))^2$, которой соответствует среднеквадратическая ошибка: $E_{x,y}(y-a(x))^2$ (это MSE). При минимизации MSE оптимальным алгоритмом будет $a = E(y|x)$. Чтобы использовать такой алгоритм нам необходимо знать распределение, но мы его не знаем $\Longrightarrow$ переходим к выбору алгоритма на основе **метода обучения**.

Из обучающей выборки генерируем выборки (произвольным способом) с фиксированным размером n. Для каждой выборки выбираем метод обучения – μ (он выбирается из некоторого семейства алгоритмов A), обучаем μ на соответствующей ему выборке и получаем предикты:

$$a(x)_{X^n} = \mu(X^n)(x) \text{ – это уже ответы}$$

Для оценки качества **метода обучения** берем среднее по всем выборкам значение среднеквадратической ошибки (усредняем MSE по всем X^n):

$R(a) = E_{X^n}(E_{x,y}(y-a(x))^2)$ – ошибка, разложение на компоненты которой хотим понять

Разложение ошибки:

$$R(a) = \underbrace{E_{x,y}(y - E(y|x))^2}_{\text{шум}} + \underbrace{E_{x,y}[(E(y|x) - E_{X^n}\mu(X^n)(x))^2]}_{\text{смещение}} + \underbrace{E_{x,y}[E_{X^n}[(\mu(X^n)(x) - E_{X^n}\mu(X^n)(x))^2]]}_{\text{дисперсия}}$$

Логика:

- Шум показывает ошибку **лучшей модели** $= E(y|x)$, это шум исходных данных, устранить не выйдет
- Смещение показывает на сколько наша модель (среднее по обученным алгоритмам $\mu(X^n)$) отличается от лучшего алгоритма $E(y|x)$
- Дисперсия показывает разброс ответов обученных алгоритмов относительно среднего ответа

Разложение верно почти для любой функции потерь (у нас была квадратичная).