Конспект по теме "Алгоритмы машинного обучения"

Линейная регрессия и функция ошибки

Линейная регрессия относится к задачам обучения с учителем: когда модель должна найти взаимосвязи между признаками объектов и значением целевой переменной. В случае регрессии целевая переменная — не значение класса, а число.

Напомним условия задачи, которую мы решаем:

`n` — количество наблюдений в наборе данных;

`k` — число признаков;

$$\vec{y} - \text{вектор (список) значений целевой переменной для наших данных}$$

`X` — матрица объект-признак (`n x k`), где строки — наблюдения, а столбцы — признаки.

В матрице каждому наблюдению соответствует вектор, он же — набор значений признаков

$$\vec{x} = (x_1, x_2, ..., x_k)$$

Матрица X выглядит так:

$$X = \begin{pmatrix} x_{1_1} & \cdots & x_{1_k} \\ \vdots & \ddots & \vdots \\ x_{n_1} & \cdots & x_{n_k} \end{pmatrix}$$

Ищем единую формулу, которая по значениям `k` признаков вычислит прогноз (оценку) для любого объекта:

$$f(\vec{x}) = f(x_1, x_2, ..., x_k) = \hat{y}$$

Если найдём такую формулу и применим ко всем объектам датасета, получим вектор (набор) прогнозов для всех объектов:

$$\vec{\hat{y}} = (\hat{y_1}, \hat{y_2}, ..., \hat{y_n}) - \text{вектор (список) прогнозов}$$

Где:

$$\hat{y_i} - \text{прогноз для конкретного наблюдения i}$$

Полученный вектор можно сравнить с исходным вектором значений целевой переменной и понять, насколько формула хороша. Например, посчитав среднюю ошибку для всех наблюдений. Сперва прикинем, как может выглядеть функция. А затем настроим её параметры. Проще всего предположить, что целевую переменную описывает линейная функция от признаков:

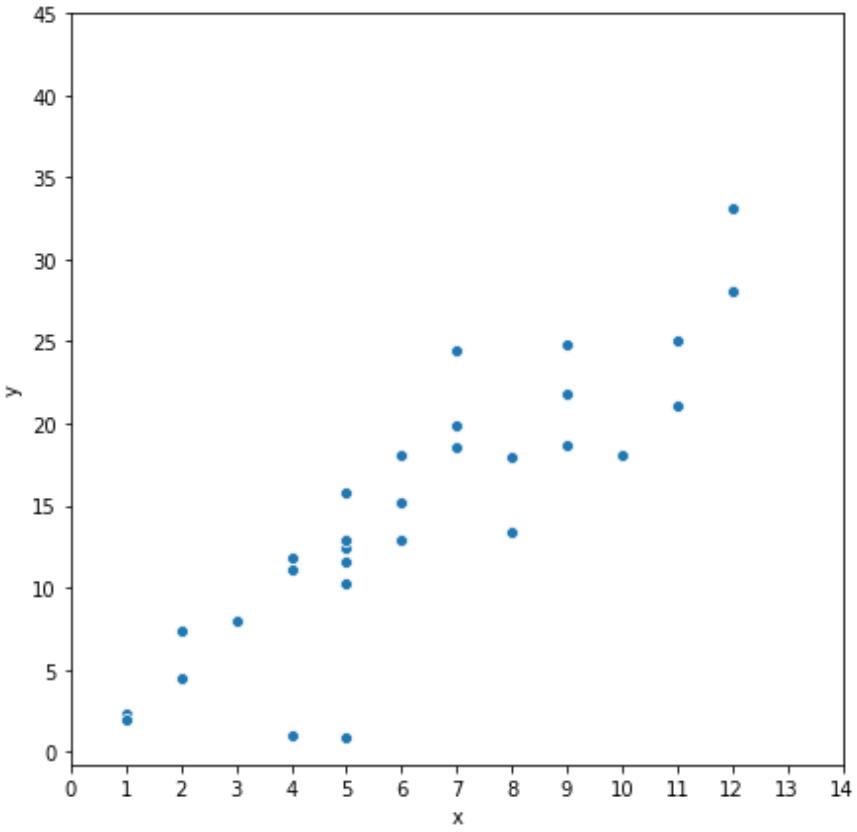
$$y(x) = w_0 + w_1 * x_1 + w_2 * x_2 + ... + w_k * x_k$$

Здесь:

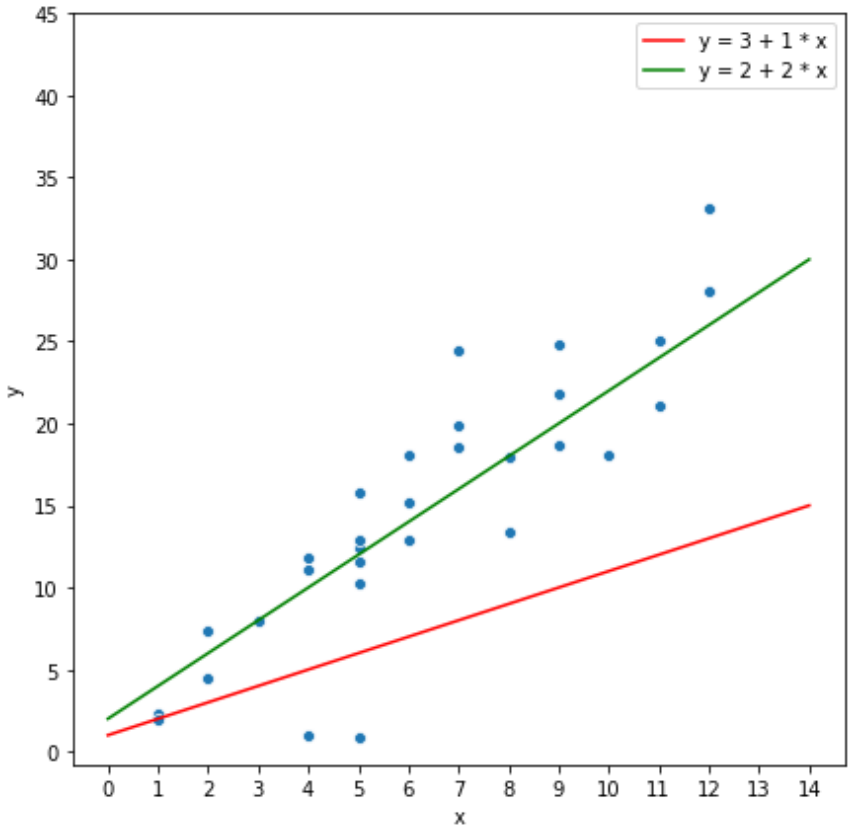
$$(w_0, w_1, w_2, ..., w_k) - \text{числовые коэффициенты}$$

Эти коэффициенты называют **веса:** они определяют вес, или вклад каждого признака в сумме, которая и определяет наш прогноз. Тогда нужно найти такие значения весов, чтобы функция давала наиболее близкий к реальным значениям результат.

Предположим, что для каждого наблюдения у нас есть всего один признак. Нарисуем для каждого наблюдения точку, где по оси X — значение этого признака, а по оси Y — значение целевой переменной:

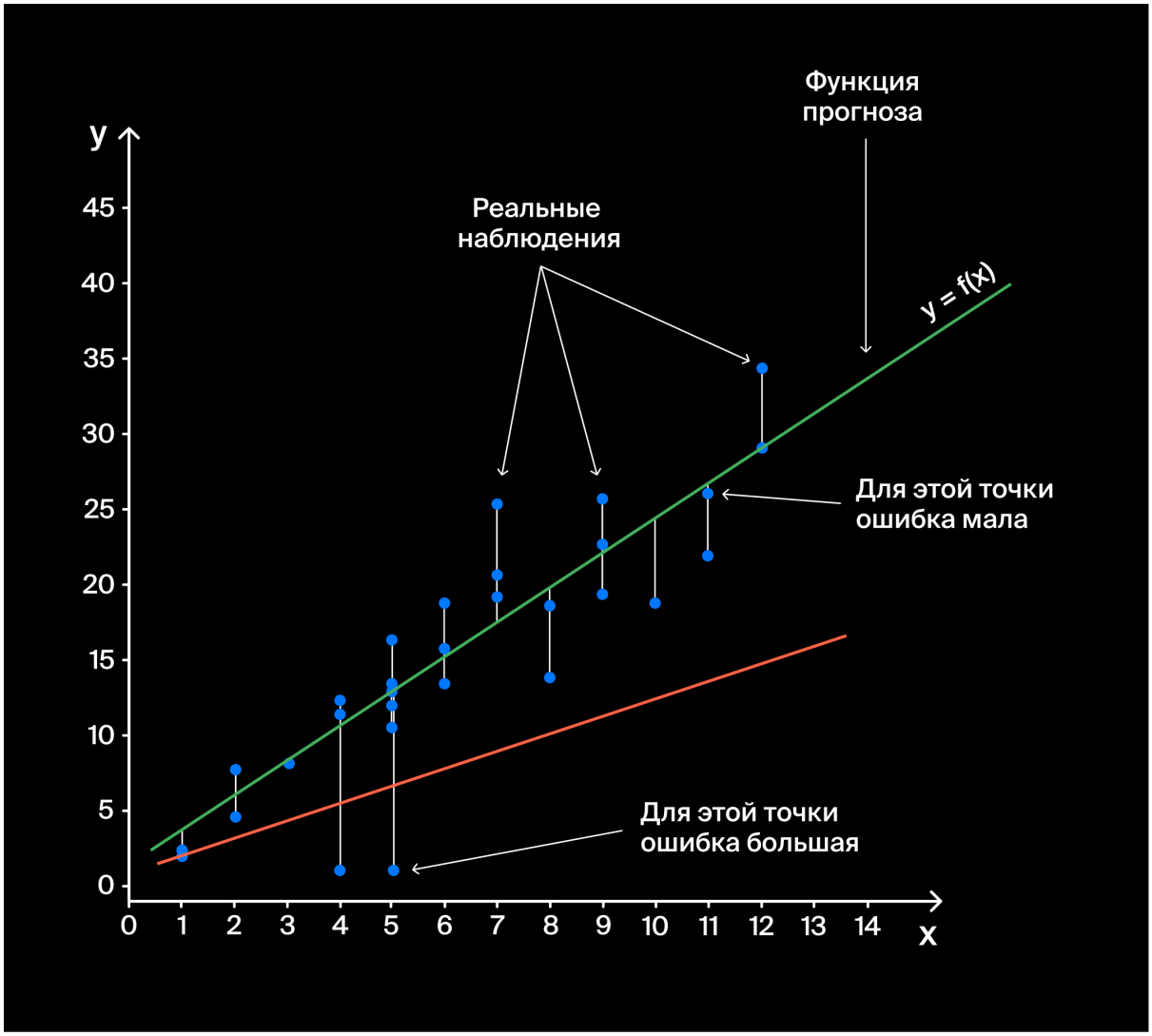


Нужно найти такую функцию вида `y = w_0 + w * x`, график которой будет прямой. Причём эта прямая должна подходить для описания наблюдений. Одни прямые лучше характеризуют набор, или **облако данных**, другие — хуже. Например, можно построить график функции `y = 3 + 1 * x` или `y = 2 + 2 * x`:



Зелёная прямая лучше описывает наблюдения, чем красная. Но как более формально описать, насколько хорошо функция отражает реальную взаимосвязь между признаками?

Проще всего оценить, **насколько сильно мы ошибаемся** в наших прогнозах в сравнении с реальным значением целевой переменной. Введем **функцию ошибки** — `Q(w)`. Она будет зависеть от выбранных нами весов.



Что считать ошибкой? Представляется, что это разница между спрогнозированным и реальным значением. Однако в чистом виде такая оценка не корректна: разница между ожиданием и реальностью может быть отрицательной, а ошибка — нет. Если прогноз идеален, то и ошибка равна нулю. А если предсказанное значение меньше или больше реального, то ошибка — положительна.

Чтобы ошибка точно была неотрицательной величиной, её возводят в квадрат:

$$(y_i - \hat{y}_i)^2$$

С одними объектами модель будет ошибаться меньше, с другими — больше. Чтобы оценить ошибку в целом, найдём среднюю ошибку на всех объектах:

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

Функцию ошибки определили. Осталось найти такие веса `w`, чтобы значение функции ошибки было минимальным.

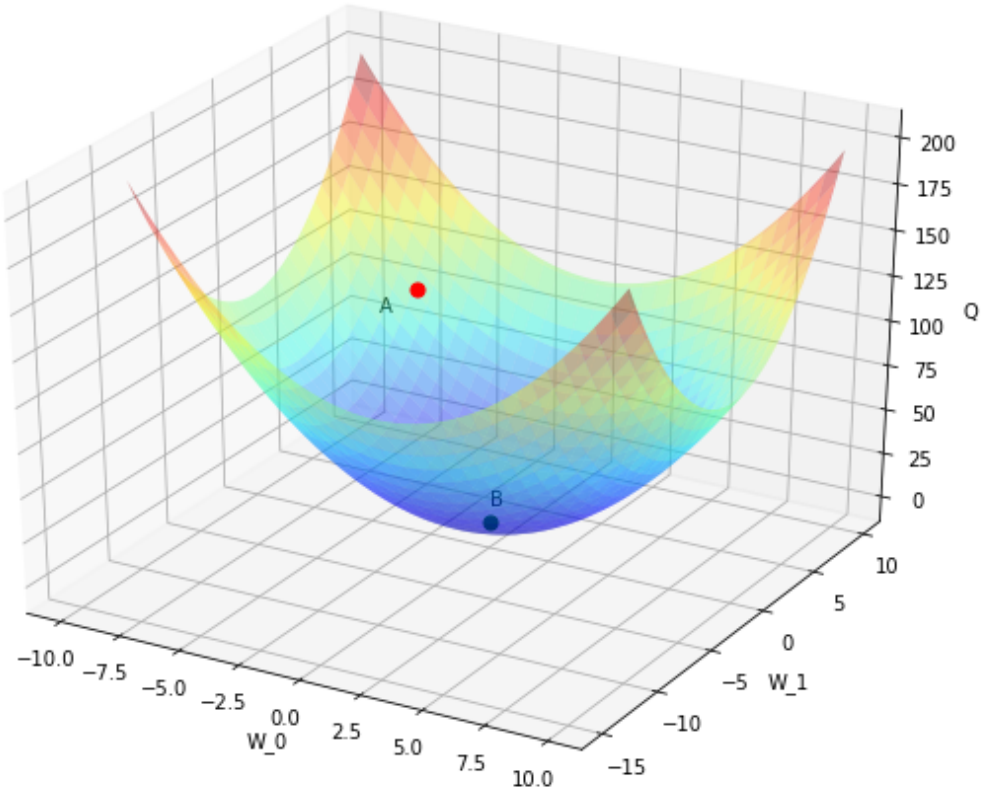
Градиентный спуск

Ошибка модели при определённом наборе весов задаётся формулой:

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

где `y(x) = w₀ + w₁ * x₁ + w₂ * x₂ + ... + w_k * x_k`

Ваша задача — найти такое сочетание весов, при котором значение этой функции будет минимальным. Вернёмся к случаю, когда есть всего один признак и два коэффициента, которые нужно подобрать: `y = w₀ + w₁ * x`. Мы можем посчитать для большого количества разных `w₀` и `w₁` соответствующие значения функции ошибки. Тогда график функции ошибки будет таким:



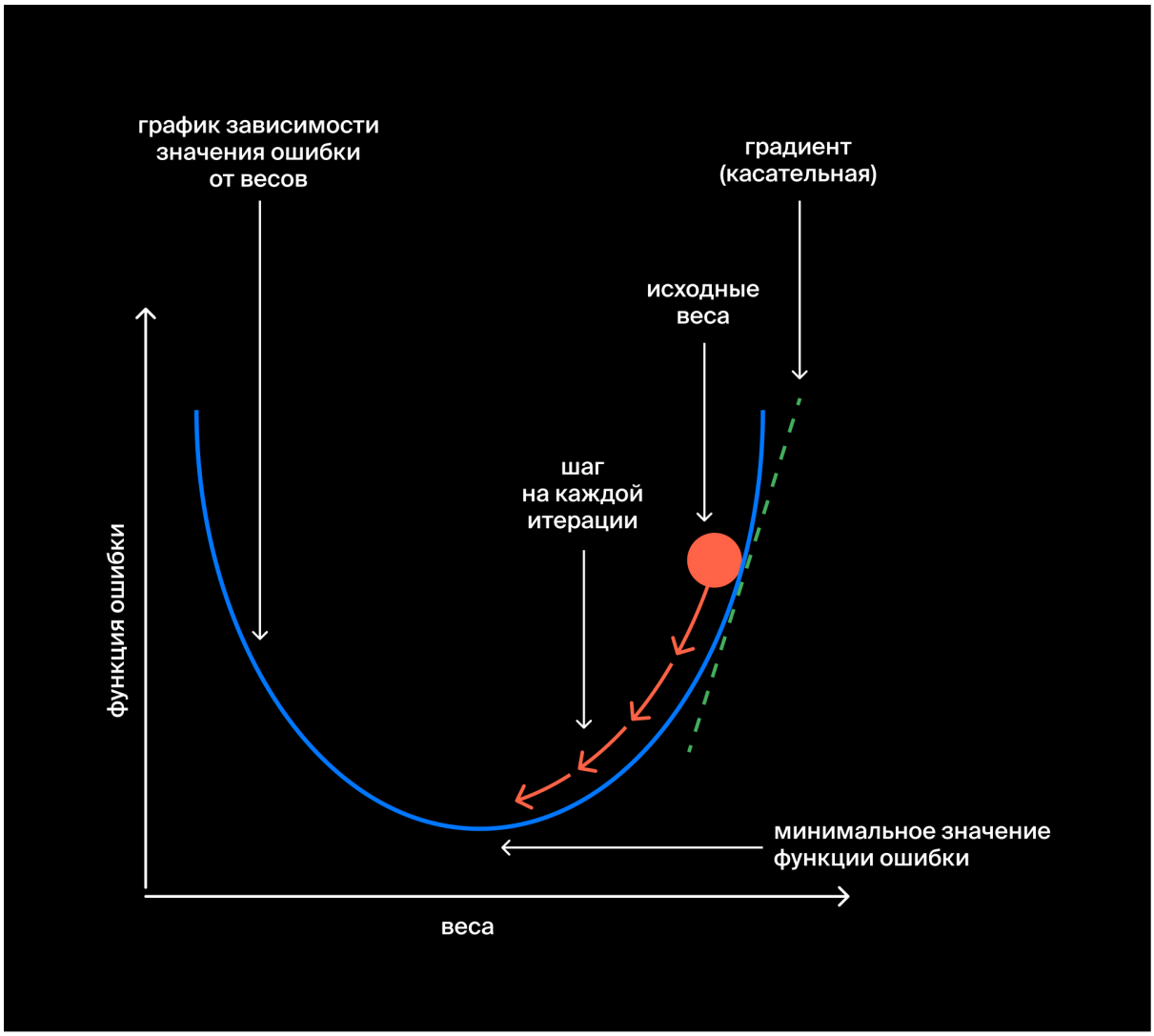
Это **график трёхмерной поверхности**. Допустим, вы выбрали значения двух коэффициентов `w_0_a` и `w_1_a` и получили значение функции `Q_a.` На рисунке этому соответствует точка A. Нам нужно прийти в точку, где значение функции минимально. Отметим её как B. Точка B соответствует тому набору весов, при котором средняя квадратичная ошибка на наблюдениях минимальна. В примере с прошлого урока можно считать, что точка A соответствует красной прямой, а точка B — зелёной.

Поиск локального минимума функции и есть задача оптимизации. В машинном обучении применяют особый метод оптимизации — **градиентный спуск.** Выбирается начальная точка (в нашем случае A) со стартовым значением весов. Её шаг за шагом немного смещают, стараясь двигаться в сторону минимума функции ошибки. Важные вопросы метода:

- Куда шагать? (направление);
- Насколько шагать? (величина шага).

Куда именно «шагнуть» из текущей точки? Интуитивно понятно, что лучше в ту сторону, где «спуск» к минимуму самый резкий. Значит, нужно

сдвинуться к «наискорейшему спуску», который соответствует максимальному изменению значения функции при определённой длине шага. Для двумерного случая направление сдвига идёт по касательной из текущей точки в сторону минимума. Если зафиксируем значение нулевого коэффициента `w_0`, тогда срез плоскости, описывающей зависимость ошибки `Q` от веса `w_1` при единственном признаке, выглядит как парабола:



Из первоначальной точки движемся вниз по направлению касательной. Касательная задаётся как производная функции — в нашем случае мы движемся вдоль производной функции ошибки по весам:

$$\frac{\partial Q}{\partial w_1}$$

Если вернуться к многомерному случаю, касательная к поверхности— вектор производных для каждого веса:

$$\nabla Q(w) = (\frac{\partial Q}{\partial w_0}, ..., \frac{\partial Q}{\partial w_k})$$

Такой вектор называют **градиент**. Он положительный. Нам же нужно двигаться вниз к минимуму функции. Значит, будем перемещаться в противоположном направлении — в сторону **антиградиента**.

Куда идти, определили. Осталось выяснить, как быстро это делать: насколько большими шагами следует двигаться «ко дну»? За это отвечает **параметр скорости обучения** — ***learning rate***. Обозначим его за `α`. Тогда корректировку весов на каждом шаге можно записать так:

$$\vec{w}_{t+1} = \vec{w}_t - \alpha \nabla Q(\vec{w}_t)$$

Шаг надо выбирать не слишком большой — чтобы он не «перепрыгивал» минимум. Но и не слишком малый — чтобы процесс обучения не был чересчур долгим. Чаще всего **гиперпараметр** — этот термин применяют, чтобы не было путаницы с параметрами функции `w` — подбирают в ходе экспериментов. А изначально задают довольно небольшим: например, 0,001 или 0,003. Здесь шаг (гиперпараметр) — это ***learning rate.***

На практике классический — ещё говорят «ванильный» (англ. *vanilla*) — градиентный спуск применяют редко. Из-за таких недостатков:

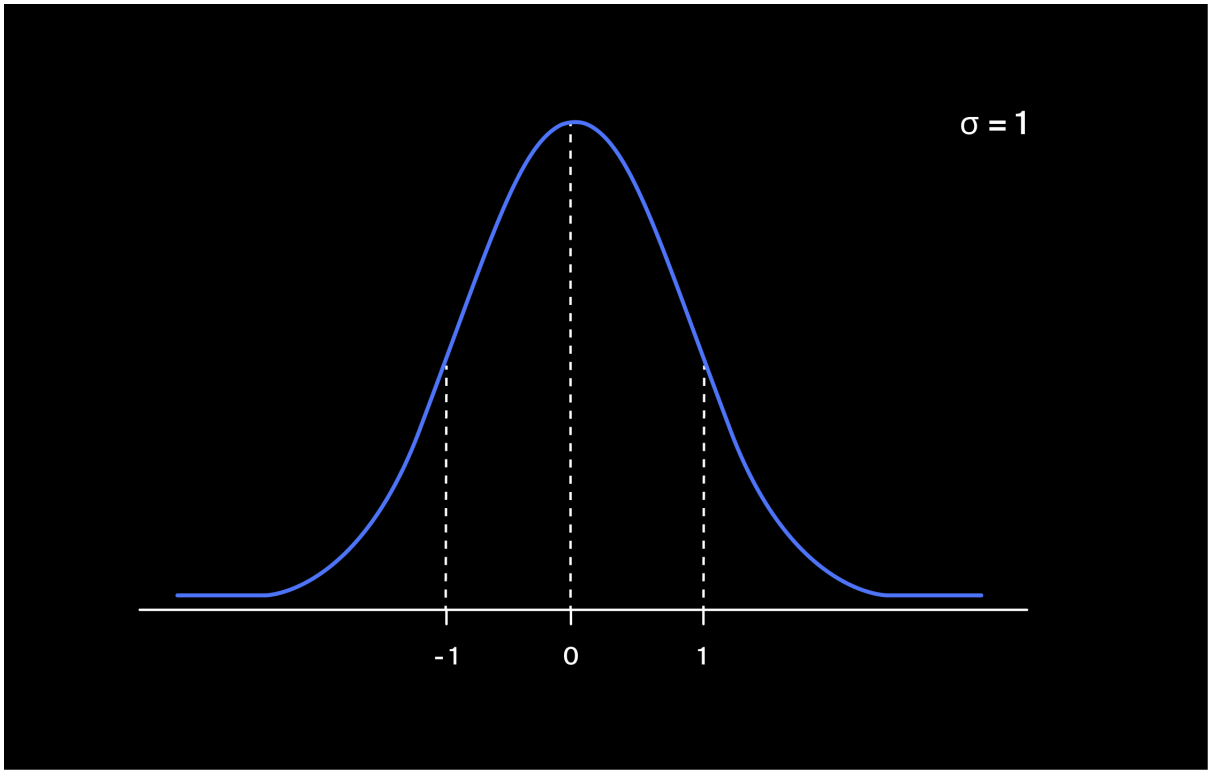
- если поверхность функции ошибки имеет много «впадин», в зависимости от начального выбора весов мы можем опуститься не в «самый глубокий овраг» — **глобальный минимум функции**, а только в один из них — **локальный минимум**. Приходится по несколько раз задавать начальные предположения.
- низкая скорость расчёта градиента по всем примерам. Чтобы ускорить сходимость и побороться с переобучением, применяют **случайный (стохастический) градиентый спуск.** При нём вычисляют градиент не на всей выборке, а на подвыборках — **мини-батчах**.
- экзотические формы поверхности функции ошибки. Тут в ход пускают не просто направления, а следующие порядки производных функции ошибки по коэффициентам.

Предобработка. Масштабирование признаков

При работе с линейной регрессией учитывайте два момента:

- признаки следует привести к стандартному виду;
- модель линейной регрессии склонна к переобучению и неустойчива в случае взаимной корреляции признаков.

Если диапазоны значений признаков сильно различаются, когда вы начнёте подбирать коэффициенты, заметите, что веса для разных признаков должны учитывать масштаб этих признаков. Так, у признаков с меньшим масштабом веса должны быть больше, а у остальных — меньше. При этом значения коэффициентов следует подобрать такими, чтобы можно было оценить вклад каждого признака в общую взвешенную сумму. Чтобы веса можно было сравнивать, признаки приводят к единому масштабу. Для этого применяют **нормализацию** — перевод значений признаков в диапазон от 0 до 1**.** Однако чаще масштабируют другим способом — **стандартизацией.** И вот почему. Минимизировать среднеквадратичную ошибку уместно, только если данные имеют вид **стандартного нормального распределения.** Математическое ожидание равно нулю, а стандартное отклонение — 1:



Стандартизация приводит значение признака именно к такому виду: для каждого наблюдения из исходного значения признака вычитается среднее, а полученная разность делится на стандартное отклонение. На практике

необязательно знать, по какой формуле исходные признаки получают значение из этих интервалов. Тем не менее, для решения задач методами линейной регрессии и не только (например, это справедливо и для кластеризации, о которой вы узнаете позже) данные обязательно стандартизируют.

В `sklearn` для нормализации и стандартизации данных в модуле `preprocessing` есть готовые классы `MinMaxScaler()` и `StandardScaler()` соответственно. Они напоминают модели: их также нужно обучать — показывать, какие значения принимает признак на примере обучающей выборки, — и только после этого применять к любым новым наблюдениям. Синтаксис работы нормализации и стандартизации одинаковый:

```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler() # создаём объект класса scaler
scaler.fit(X) # обучаем стандартизатор
X_sc = scaler.transform(X) # преобразуем набор данных
```

Следуя стандартному пайплайну машинного обучения, вы разбиваете выборку на обучающую и валидационную. Чтобы стандартизировать признаки, нужно знать их среднее и стандартное отклонение по наблюдениям. Однако вы не можете предсказать значения признаков в будущем. На валидации это надо иметь в виду.

Поскольку модель, обучаясь на *train*-выборке, видит только среднее и стандартное отклонение на ней, стандартизируют так:

- рассчитывают среднее и стандартное отклонение на обучающей выборке (обучив `StandardScaler()`) и преобразуют данные;
- применяют уже обученный `StandardScaler()` к валидационной выборке (с учётом среднего и отклонений из обучающей);
- обучают модель на стандартизированных данных из обучающей выборки;
- делают прогноз для стандартизированных данных на валидационной выборке.

Регуляризация

Простые линейные модели нетрудно реализовать и легко интерпретировать. Однако и они дают сбои. Например, из-за **мультиколлинеарности**. Мультиколлинеарность возникает, когда есть группа линейно зависимых признаков — взаимозависимых или очень сильно скоррелированных. Скажем, если в датасет «просочились» два признака, отвечающие за одно и то же расстояние: один — в метрах, другой — в километрах. Или признаки просто очень связаны — как температура воды в пруду и температура воздуха. Если коэффициент корреляции между двумя признаками слишком большой (часто больше 0.8), с линейной регрессией возникнут проблемы. А именно — нерепрезентативные веса и переобученная под эти признаки модель. Почему?

Вот наглядный пример. Два признака совпадают — предельный случай высокой взаимной корреляции:

$$x_1 = x_2$$

А теперь взгляните на три уравнения:

$$y = w_0 + w_1 * x_1 + w_2 * x_2$$
$$y = w_0 + (w_1 + w_2) * x_1 + 0 * x_2$$
$$y = w_0 + 100000 * (w_1 + w_2) * x_1 - 99999 * (w_1 + w_2) * x_2$$

Все они дадут одинаковый результат! И ошибки, вычисленные этими моделями для прогнозов относительно факта, тоже одинаковы. Алгоритм не может решить, какой из трёх векторов весов лучше. Таких равноправных векторов бывает бесконечное множество. По ним алгоритм не оценит адекватно влияние отдельного признака на целевую переменную.

Как с этим бороться? Оставить только признаки, корреляция между которыми не превышает высокого порога (скажем, 0.8). Например, вычислить матрицу корреляций методом `corr()` и для каждой пары сильно скоррелированных признаков вручную удалить один:

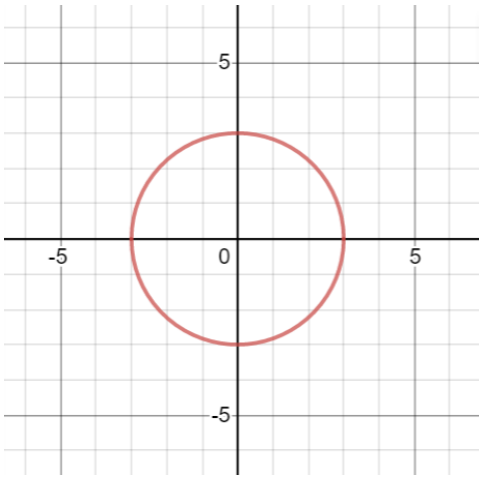
```
cm = df.corr() # вычисляем матрицу корреляций
```

Кроме ручного удаления скоррелированных признаков, есть **регуляризация.** В общем смысле это любое дополнительное ограничение

на модель или какое-то действие, которое позволяет снизить её сложность и влияние эффекта переобучения. У линейных моделей регуляризация — ограничение веса. Напрямую зададим условия так:

$$\sum_{i=1}^{m} w_i^2 < b$$

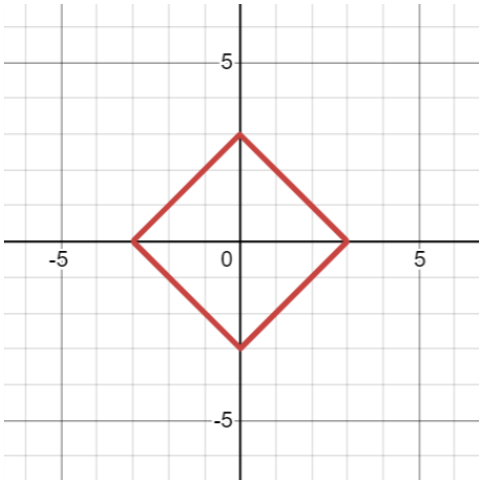
Похожее ограничение для двух признаков и b=3 выглядело бы так:



Ещё один способ ограничить масштаб весов — ограничить их модуль:

$$\sum_{i=1}^{m} |w_i| < a$$

Похожее ограничение для двух признаков и для a=3 выглядело бы так:



Часто подобные ограничения задают прямо через функцию ошибки. Так мы стараемся минимизировать ошибку при наименьших (по сумме модулей или по сумме квадратов) весах признаков. Иначе говоря, запрещаем модели подбирать признакам слишком уж большие веса, чтобы она не перекалибровывалась под мультиколлинеарные признаки. Ограничить можно двумя способами.

L1-регуляризация, или **Lasso-регрессия**, обозначается формулой:

$$Q_{Lasso} = \frac{1}{n} * (\sum_{i=1}^{n} (y_i - \hat{y_i})^2 + \lambda * ||w||_1)$$

где Q_{Lasso} – функция ошибки,
y_i – реальное значение целевой переменной для i объекта,
$\hat{y_i}$ – прогноз для i объекта,
λ – коэффициент регуляризации,
$||w||_1$ – сумма модулей весов

(т. е. минимизируя ошибку подбором оптимальных весов, дополнительно стараемся ещё и минимизировать **сумму модулей** этих весов)

L2-регуляризация, или **Ridge-регрессия**:

$$Q_{Ridge} = \frac{1}{n} * (\sum_{i=1}^{n} (y_i - \hat{y_i})^2 + \lambda * ||w||_2^2)$$

где Q_{Ridge} – функция ошибки,
y_i – реальное значение целевой переменной для i объекта,
$\hat{y_i}$ – прогноз для i объекта,
λ – коэффициент регуляризации,
$||w||_2^2$ – сумма квадратов весов

(т. е. минимизируя ошибку подбором оптимальных весов, дополнительно стараемся ещё и минимизировать **сумму квадратов** этих весов)

В `sklearn` есть несколько алгоритмов линейной регрессии, которые уже включают в себя регуляризацию:

- **Lasso regression** использует L1-регуляризацию. Этот алгоритм «зануляет» веса при всех сильно скоррелированных признаках, кроме одного. Таким образом, в алгоритм встроен механизм **отбора**

признаков (feature selection) — его применяют, когда нужно снизить размерность и избавиться от дублирующих признаков.

- **Ridge regression** использует L2-регуляризацию. В этом случае веса между скоррелированными признаками будут распределены примерно равномерно.

Реализация линейных моделей

В библиотеке `sklearn` алгоритмы линейной регрессии находятся в модуле `linear_model`.

- `linear_model.LinearRegression()` — модель стандартной линейной регрессии. Перед тем как её применять, нужно: стандартизировать данные и вручную провести отбор признаков, чтобы избавиться от мультиколлинеарности.
- `linear_model.Lasso()`: модель линейной регрессии со встроенной *L1*-регуляризацией весов (ограничение на сумму модулей весов). Она также требует стандартизации данных, но за отбор признаков отвечает встроенная регуляризация. Специально избавляться от сильно скоррелированных признаков в этом случае уже не нужно.
- `linear_model.Ridge()`: модель линейной регрессии со встроенной *L2*-регуляризацией весов (ограничение на сумму квадратов весов). И тут не обойтись без стандартизации данных, чтобы свести признаки к одному масштабу. В случае с *Ridge*-регрессией, как и в *Lasso*, необязательно очищать признаки от коррелированных. Сама по себе *Ridge*-регрессия не сделает отбор признаков за вас: она не «зануляет» веса дублирующих признаков. Функция регуляризации устроена так, что между похожими признаками модель делит веса примерно поровну.

Синтаксис каждой функции можно посмотреть в официальной документации: https://scikit-learn.org/stable/modules/generated/sklearn.linear_model

В случае задачи линейной регрессии обучают модель и строят прогноз стандартными методами `fit()` и `predict()`. Например, для Ridge-регрессии это будет выглядеть так:

```
from sklearn.linear_model import Ridge

model = Ridge() # создаём модель класса Ridge-регрессия
model.fit(X_train, y_train)
y_pred = model.predict(X_val)
```

Алгоритмы линейной регрессии отличаются тем, что дают возможность интерпретировать веса для каждого признака. После стандартизации значения коэффициентов при признаках отражают степень влияния каждого на финальный прогноз. Чем больше по модулю коэффициент, тем выше его влияние. Обучив модель, вы сможете напрямую вывести все коэффициенты, которые соответствуют финальной, оптимальной функции. Веса при признаках выводят методом `.coef_` модели:

```
print(model.coef_)
```

...а значение нулевого коэффициента (англ. *intercept*) — методом `.intercept_`.

```
print(model.intercept_)
```

Метрики регрессии

Метрики смотрят на валидационной выборке — проверяют, насколько хорошо работает модель на тех данных, которые она не видела при обучении. Это моделирует реальную ситуацию: когда модель придётся применять, не зная правильных ответов.Все формулы расчёта метрик приводятся для данных из валидационной выборки.

Главное в задаче регрессии — приблизиться насколько можно к реальному значению.

Пусть:

y_i – реальное значение целевой переменной для конкретного наблюдения i,

$\hat{y}_i$ – прогноз для конкретного наблюдения i,

n – число наблюдений в валидационной выборке

Рассмотрим наиболее популярные в бизнесе метрики:

- Средний модуль ошибки — **MAE** (англ. *Mean Absolute Error*);
- Средняя квадратичная ошибка — **MSE** (англ. *Mean Squared Error*) и корень из этой величины — **RMSE** (англ. *Root Mean Squared Error*);
- Коэффициент детерминации или R-квадрат (**R2**);
- Средняя относительная ошибка по модулю — **MAPE** (англ. *Mean Absolute Percentage Error*).

MAE

Название метрики объясняет способ её расчёта. Сперва оцениваем, насколько наши прогнозы отклоняются от реальных значений целевой переменной по модулю. Потом усредняем модуль ошибки по всем наблюдениям из валидационной выборки. Разницу берут по модулю, и оттого неважно, отклоняется прогноз в большую или в меньшую сторону:

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

MSE и RMSE

Можно сделать так, чтобы метрика отражала чувствительность модели к выбросам. Если для каких-то значений модель ошибается сильнее, ошибку следует «усилить». Тогда считают средний квадрат ошибки (MSE):

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

... или RMSE — это квадратный корень из MSE:

$$RMSE = \sqrt{\frac{1}{n} * \sum_{i=1}^{n} (y_i - \hat{y_i})^2}$$

Коэффициент детерминации или R-квадрат (R2)

Эта знакомая вам метрика показывает, какую долю изменчивости целевой переменной уловила модель. Формула коэффициента детерминации:

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

, где:

$$\bar{y} = \frac{1}{n} * \sum_{i=1}^{n} y_i$$

Коэффициент детерминации принимает значения от 0 до 1. Чем он ближе к единице, тем лучше ваш прогноз соотносится с реальными значениями целевой переменной.

MAPE

Относительная ошибка бывает важнее абсолютных значений (как MAE, MSE и RMSE). Скажем, `MAE=42` — это много или мало? Если среднее значение вашей величины 56, отклонение 42 более чем существенно и, кажется, ваша модель не работает. А если целевая переменная где-то в диапазоне от 3000 до 4000, то 42, напротив, очень хороший показатель (если не «подозрительно хороший»). Поэтому среднюю относительную ошибку оценивают, вычисляя по формуле:

$$MAPE = 100\% * \frac{1}{n} * \sum_{i=1}^{n} \frac{|y_i - \hat{y}_i|}{|y_i|}$$

MAPE — интуитивно более понятная оценка. Средняя относительная ошибка в 2% внушает оптимизм. А вот если это 95%, нужно или модель переделывать, или в расчётах ошибку искать, а возможно, и другие данные. И всё же неопределённсть никуда не исчезает: где-то хватает точности в 80%, а в других случаях мало даже 95%.

Метрики и функция ошибки

При поиске оптимальной функции все алгоритмы стандартных библиотек Python, подбирая веса, минимизируют функцию потерь. Она часто совпадает с той метрикой, которую вы оцениваете на валидационной выборке. Например, многие алгоритмы по умолчанию в качестве функции потерь берут MSE. Но иногда дефолтная функция потерь и интересующая вас метрика не совпадают: тогда алгоритм будет «настраиваться» не под то, что вам нужно.

Почти во всех реализациях алгоритмов вы можете напрямую задать ту функцию, которая будет оптимизироваться уже при обучении. Если хотите сопоставить функцию потерь с той метрикой, которую будете оценивать на валидации, читайте документацию.

Расчёт метрик в *sklearn*

Разработчики `sklearn` создали модуль `metrics`, где отразили некоторые метрики:

- MAE — `metrics.mean_absolute_error`
- MSE — `metrics.mean_squared_error`
- R2 — `metrics.r2_score`

Метрик RMSE и MAPE в этом модуле нет. Однако вы можете написать соответствующие функции сами.

Разберём синтаксис работы с методами метрик на примере функции `metrics.mean_absolute_error`:

```
from sklearn.metrics import mean_absolute_error

print('MAE:', mean_absolute_error(y_true, y_pred))
```

В качестве параметров подают вектор истинных значений целевой переменной — `y_true` и вектор спрогнозированных значений — `y_pred`. Метрики `metrics.mean_squared_error` и `metrics.r2_score` вычисляют аналогично.

Реальные данные, которые попадают на вход модели после её разработки, называют **тестовая выборка** (англ. *test data*). Однако перед тем как оценивать качество модели в бою, аналитики проверяют её работу на **валидационной выборке** (англ. *validation data*).

Логистическая регрессия

Это алгоритм для решения задачи бинарной классификации. Напомним, бинарная классификация — это частный случай классификации, когда классов всего два: `"0"` или `"1"`. Говорят, что целевая переменная здесь — **бинарная величина**.

Модели, обученные алгоритмами для бинарной классификации, могут не просто прогнозировать финальное значение класса для какого-то объекта или клиента, а ещё и оценивать вероятность рассматриваемого события.

Популярный алгоритм для решения таких задач — **логистическая регрессия**. Она реализована как класс `LogisticRegression()` в том же модуле `linear_model` библиотеки `sklearn`, что и алгоритм линейной регрессии:

```
from sklearn.linear_model import LogisticRegression

model = LogisticRegression()
```

Разберём, какая теория стоит за работой этого алгоритма. Ваша функция должна возвращать вероятность наступления какого-то события. Допустим, она рассчитывается по формуле:

$$f(z) = \mathbb{P}\{y = 1 \mid x\}$$

Это уравнение значит «вероятность того, что y=1 при условии, что признаки принимают значения x, выражается функцией f от переменной z»,

где z — линейная функция от значений признаков (вектора x), как в случае с линейной регрессией:

$$z(x) = w_0 + w_1 * x_1 + w_2 * x_2 + ... + w_n * x_n$$

В алгоритме логистической регрессии предполагается, что f — **логистическая функция**. Она имеет вид:

$$f(z) = \frac{1}{1 + e^{-z(x)}}$$

График функции выглядит так:

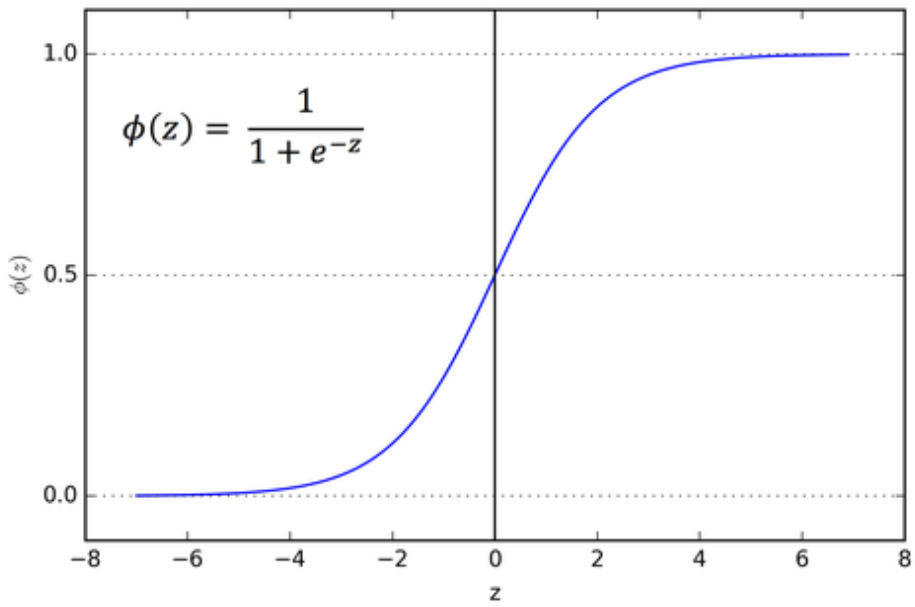


График логистической функции называют **сигмоида**. По нему видно, что логистическая функция принимает значения от 0 до 1. Внешний вид и область допустимых значений сигмоиды хорошо подходит для представления вероятности наступления какого-то события. Эта кривая «загибается» на бесконечно маленьких и бесконечно больших значениях `z`. Если уже взвешенная сумма принимает достаточно высокие значения, при которых вероятность наступления события, допустим 0.95, дальнейшее увеличение этих признаков уже не так сильно повышает и без того высокую вероятность, и с ростом признаков она стремится к 1. То же справедливо и для малых значений вероятности, но в обратную сторону.

Затем алгоритм (например, тем же градиентным спуском) так подбирает коэффициенты `w` для функции `z(x)`, чтобы `f(z)` снова принимала наиболее близкие к реальным значениям целевой переменной ответы.

По сути, когда мы решаем задачу бинарной классификации, обращаясь к логистической регрессии, то преобразуем линейную регрессию для вычисления вероятности принадлежности к классу 1. Именно поэтому логистическая, как и линейная регрессия (и её вариации: *Lasso* и *Ridge*), принадлежит к линейным моделям. Она находится в том же модуле `sklearn linear_model`.

Чтобы рассчитать прогноз класса после обучения такой модели, также применяют метод `predict()`:

```
y_pred = model.predict(X_test)
```

А чтобы получить вероятность принадлежности объекта к первому или второму классу — метод `predict_proba()`:

```
y_probas = model.predict_proba(X_test)
```

`y_probas` представляет собой двумерный массив, где каждому объекту из валидационной выборки соответствуют 2 значения: вероятность (а на самом деле, скорее уверенность) принадлежности к классу 0 («событие не наступит») и вероятность принадлежности к классу 1 («событие наступит»). В сумме для каждого объекта эти два числа дают единицу.

Когда мы обучаем модель и «подгоняем» веса на обучающей выборке, на каждом шаге проверяем, насколько близок ответ с подобранными весами тому, что есть на самом деле. Так, ещё на этапе обучения модели, можно оценить ошибку — говорят, «**оценить ошибку на обучении**». Подобрать веса для линейной модели или другого алгоритма так, чтобы для *каждого* наблюдения был правильный ответ, почти никогда не выходит. На части наблюдений всё равно получим ошибку. Её называют **ошибка на обучающей выборке**. В процессе обучения мы стараемся эту ошибку **минимизировать**. Если сделать это не удаётся и, скажем, сколько бы ни подбирали алгоритм, он работает лишь для половины наблюдений, значит модель **недообучилась**. Возникающую в таких случаях ошибку называют **ошибка смещения** (англ. *bias*, «смещение»): ответы функции *смещены* относительно данных, потому что она не улавливает всех взаимосвязей. Вот отчего так бывает:

- Слишком мало примеров или признаков;
- Слишком простая функция;
- Неверный подход к подбору разных вариантов искомой зависимости;

Такая модель будет давать одинаково плохие результаты и на обучающей, и на тестовой выборке.

В идеальном случае модель (функция, алгоритм) должна не только редко ошибаться на обучении, но и хорошо работать на новых данных, которые она не «видела» при подгонке весов или поиске оптимальной зависимости. Говорят, что модели нужно иметь хорошую **обобщающую способность**. Тогда от применения машинного обучения будет польза.

Представим модель, которая идеально точно предсказывает значения на тестовой выборке. Можно сказать, что ваша модель «идеальна», но лишь до тех пор, пока ей на вход не передадут значения признаков для новых объектов. Когда на валидационных данных модель даёт результаты значительно хуже, чем на обучающих, говорят, что она **переобучилась** (англ. *overfitting*). Такую ошибку называют **ошибка разброса** (англ. *variance*). Это значит, что модель слишком сильно подстраивается под данные с учётом их шума: при переобучении она учитывает не только действительные связи в данных, но и избыточную информацию.

Метрики классификации. Работа с метками

Метрики классификации на основе значений прогнозного класса

Сначала рассмотрим метрики, которые берут в расчёт только итоговое спрогнозированное значение — 0 или 1:

- **Матрица ошибок** (англ. *confusion matrix*)
- **Доля правильных ответов** (англ. *accuracy*)
- **Точность** (англ. *precision*) и полнота (англ. *recall*)
- **F1_score**

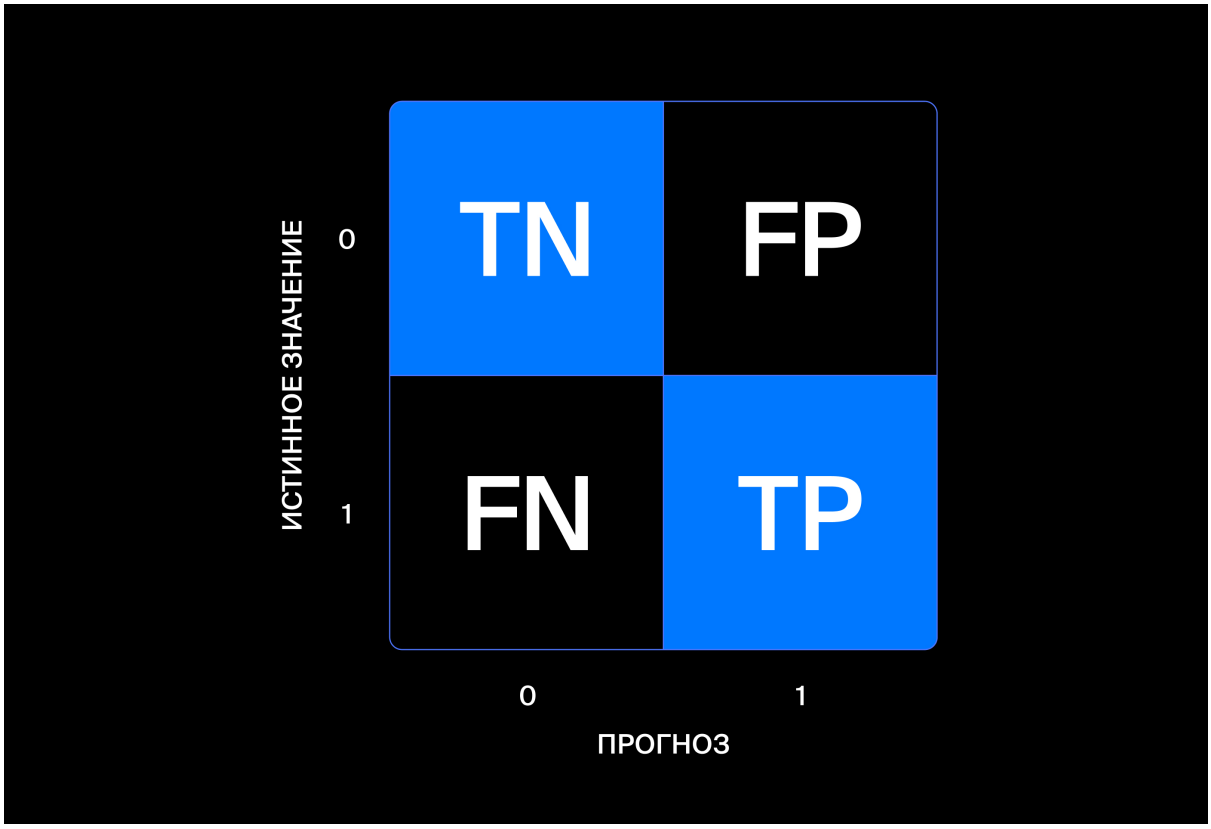
Матрица ошибок

Возможные значения классов: 0 и 1. И ваша модель тоже может выдавать итоговый прогноз в виде значения одного из двух классов. Тогда для каждого объекта прогноз относится к одной из четырех групп:

- Прогноз модели = 1, реальное значение = 1. Такие прогнозы называют **True Positive** («истинно положительные») — сокращённо **TP**.
- Прогноз модели = 1, реальное значение = 0. Такие прогнозы называют **False Positive** («ложно положительные») — сокращённо **FP**.
- Прогноз модели = 0, реальное значение = 1. Такие прогнозы называют **False Negative** («ложно отрицательные») — сокращённо **FN**.
- Прогноз модели = 0, реальное значение = 0. Такие прогнозы называют **True Negative** («истинно отрицательные») — сокращённо **TN**.

У хорошей модели бóльшая часть прогнозов должна попадать в группы TP и TN.

Матрица ошибок отражает количество наблюдений в каждой группе. Выглядит она так:



Расчёт матрицы ошибок реализован в `sklearn` в модуле `metrics`. Можно задать и переменные TN, FP, FN, TP:

```
from sklearn.metrics import confusion_matrix

cm = confusion_matrix(y_true,y_pred)
tn, fp, fn, tp = cm.ravel() # "выпрямляем" матрицу, чтобы вытащить нужные значения
```

Функция `confusion_matrix()` возвращает матрицу вида `[[TP, FP], [FN, TP]]`. Метод `.ravel()` позволяет преобразовать матрицу к сводному (одномерному) списку `[TP, FP, FN, TP]`.

По матрице ошибок вы можете понять, как именно склонен ошибаться ваш алгоритм. Есть ли у него перекос в сторону позитивного класса (чрезмерный оптимизм, слишком много FP). Или наоборот, он перестраховывается, и увеличена группа FN, как у отпетых пессимистов.

Для остальных метрик итогового прогноза классов справедливы термины, введённые для матрицы ошибок.

Доля правильных ответов

Долю правильных ответов вычисляют так:

$$Accuracy = \frac{TP + TN}{n} = \frac{TP + TN}{TP + TN + FP + FN}$$

Это доля верно угаданных ответов из всех прогнозов. Чем ближе значение *accuracy* к 100%, тем лучше.

Метрику рассчитывают функцией `accuracy_score` из модуля `metrics`. На вход функция принимает верные и спрогнозированные значения классов на валидационной выборке:

```
acc = accuracy_score(y_true, y_pred)
```

accuracy работает не всегда, а только при условии **баланса классов** — когда объектов каждого класса примерно поровну, 50% : 50%.

Точность (*precision*) и полнота (*recall*)

Чтобы оценить модель без привязки к соотношению классов, рассчитывают метрики:

$$Precision = \frac{TP}{TP + FP}$$

$$Recall = \frac{TP}{TP + FN}$$

precision говорит, какая доля прогнозов относительно `"1"` класса верна. То есть смотрим **долю правильных ответов только среди целевого класса.** В бизнесе метрика *precision* нужна, если каждое срабатывание (англ. *alert*) модели — факт отнесения к классу `"1"` — стоит ресурсов, а вы не хотите, чтобы модель часто «срабатывала попусту».

Вторая метрика нацелена на минимизацию противоположных рисков — *recall* показывает, **сколько реальных объектов** `"1"` класса **вы смогли обнаружить** с помощью модели.

Каждая метрика принимает значения от 0 до 1. Чем ближе к единице, тем лучше. Однако при настройке параметров модели — обычно порога вероятности, после которого мы относим объект к классу `"1"` — оптимизация одной метрики часто приводит к ухудшению другой. Настраивая параметры модели, вы балансируете между этими двумя показателями. Окончательное решение принимайте, исходя из целей работы.

Метрики точности и полноты также реализованы в модуле `metrics` в функциях `precision_score` и `recall_score`; имеют схожий с другими метриками синтаксис:

```
precision = precision_score (y_true, y_pred)
recall = recall_score (y_true, y_pred)
```

Обе функции возвращают число от 0 до 1.

F1-мера

Так как *precision* и *recall* направлены на избежание противоположных рисков, нужна сводная метрика, учитывающая баланс между метриками. Это **F1-score**:

$$F1 = \frac{2 * precision * recall}{precision + recall}$$

В `sklearn.metrics` F1-меру вычисляют методом `f1_score`:

```
f1= f1_score(y_true, y_pred)
```

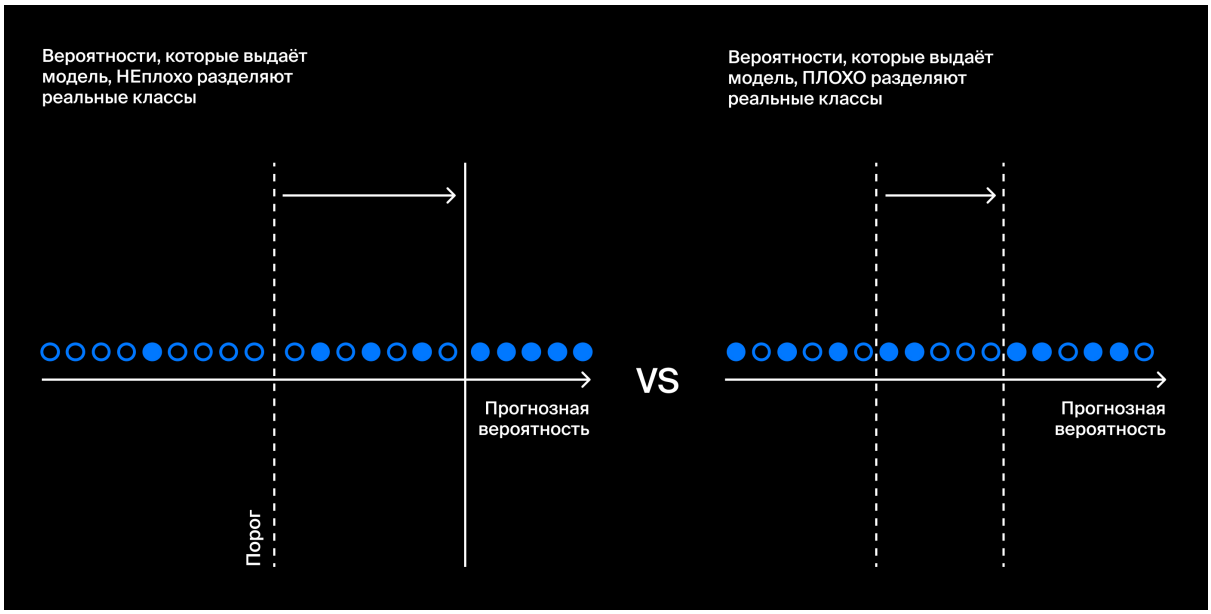
Функция также возвращает одно число от 0 до 1. Чем ближе к единице, тем лучше.

Метрики классификации. Смотрим на вероятности

Можно предсказывать не только ответ для каждого наблюдения – `"0"` или `"1"` – но и вероятность того, что наблюдение принадлежит к классу `"0"` или `"1"`. Вот почему для оценки качества **классификатора** (модели

классификации) применяют метрику **roc_auc**, или **площадь под кривой ошибок — AUC-ROC.**

Логика метрики: «Если отсортировать объекты по вероятности, которую спрогнозировала модель, насколько хорошо будут выделяться реальные классы объектов?» Иначе говоря, будет ли такое, что на одном конце в основном скопились объекты класса `"0"`, а на другом — класса `"1"`. Причём, если вероятности совсем плохо сортируют наши объекты, каким бы мы ни выбрали порог, остальные метрики лучше не станут. А когда объекты отсортированы неплохо, можно считать, что в целом модель ведёт себя адекватно, а порог настраивать уже отдельно.



Именно из-за этого свойства метрика площади под кривой ошибок — AUC-ROC — так хорошо подходит даже для несбалансированных классов.

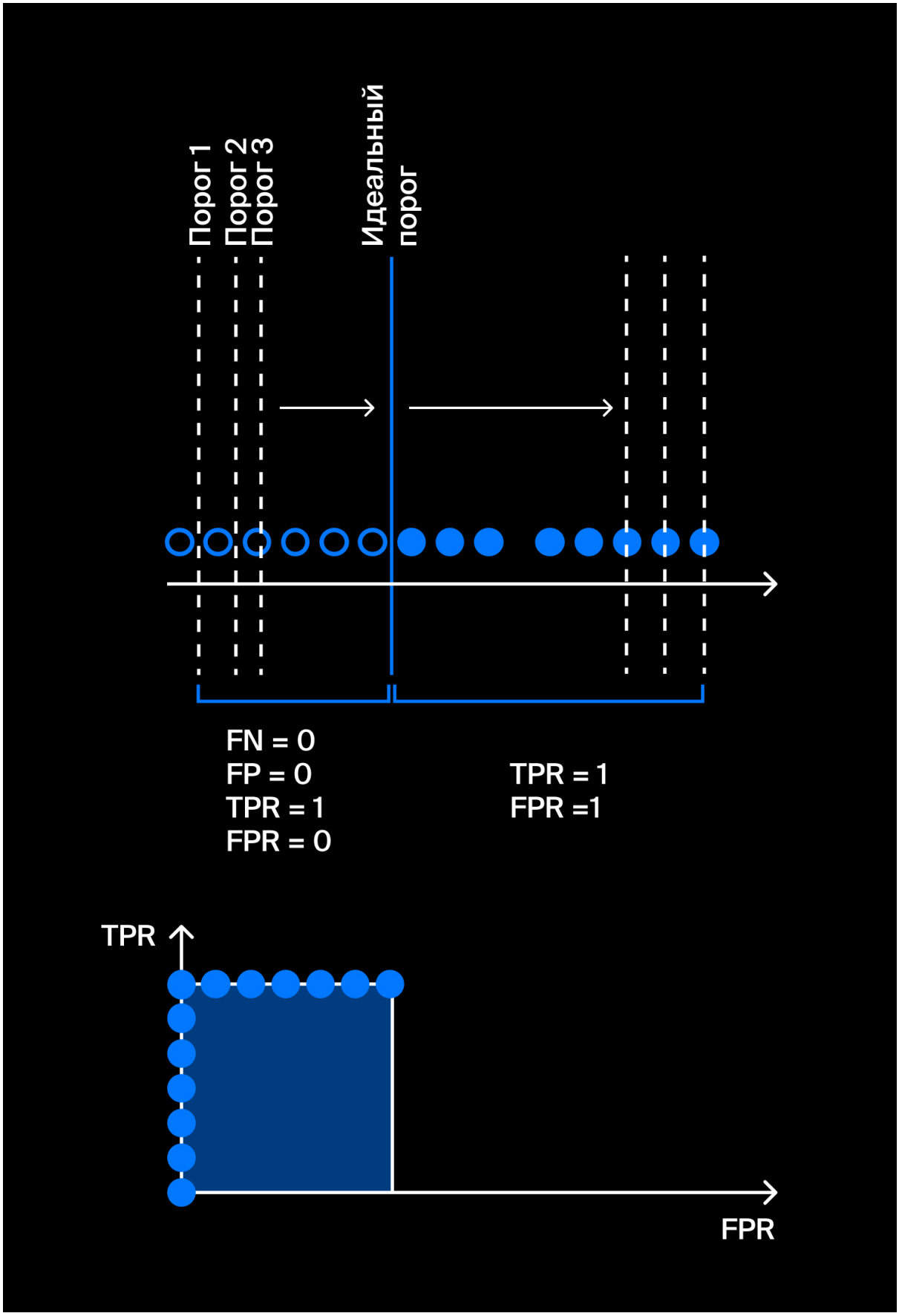
Разберёмся с ней подробнее. Введём понятия **True Positive Rate** (TPR соответствует полноте) и **False Positive Rate** (FPR — та часть, которую алгоритм ошибочно отнёс к `"1"` классу вместо `"0"`):

$$TPR = \frac{TP}{TP + FN}$$

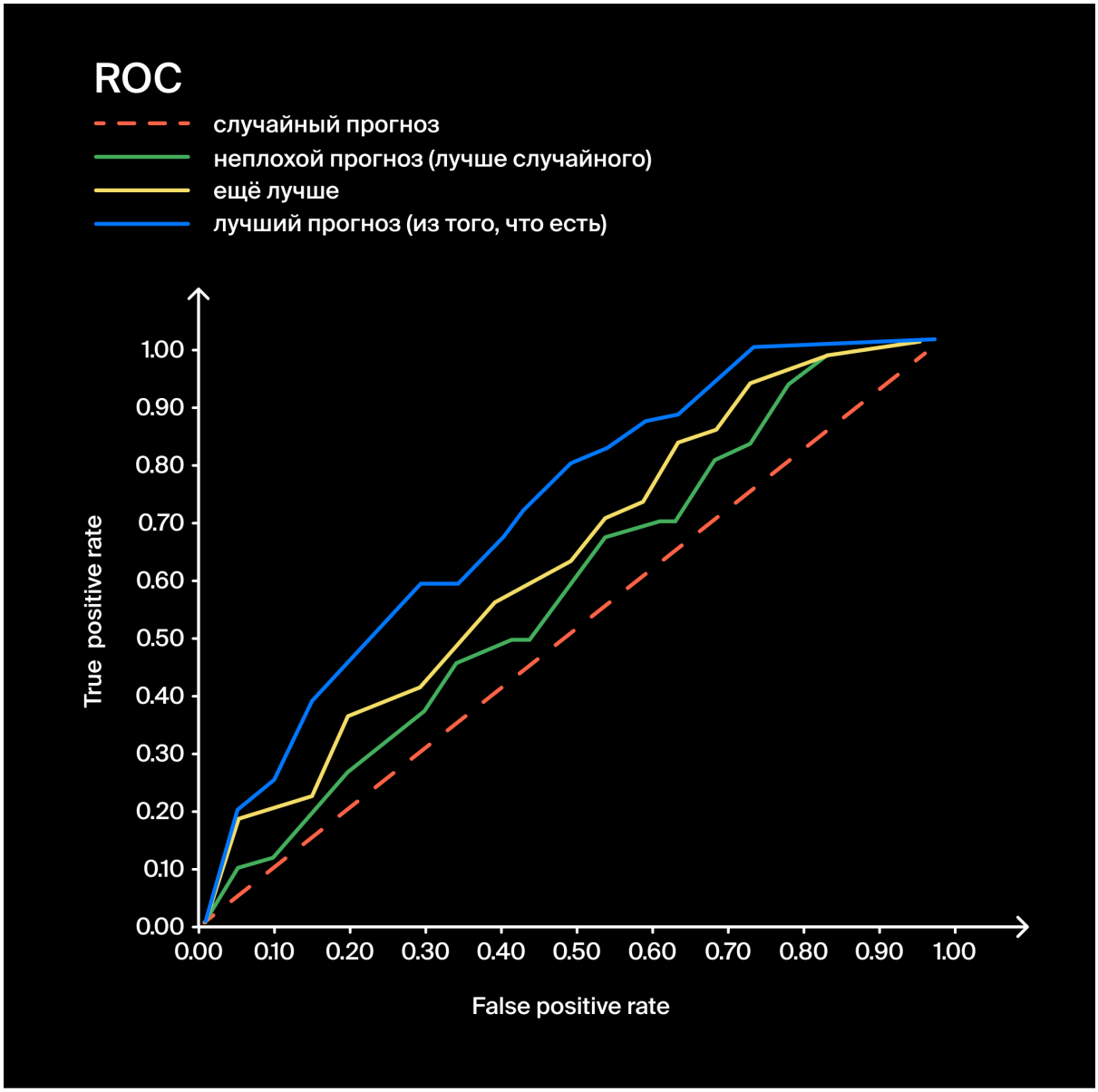
$$FPR = \frac{FP}{FP + TN}$$

Отсортируем объекты по росту прогноза вероятности и будем итеративно выбирать порог отнесения к классу, останавливаясь на каждом

наблюдении и вычисляя при таком пороге TPR и FPR. У идеальной модели объекты выглядели бы так:


Порог 1
Порог 2
Порог 3
Идеальный порог
FN = 0
FP = 0
TPR = 1
FPR = 0
TPR = 1
FPR =1
TPR
FPR

На практике не совершенны ни данные, ни модели. Идеальной сортировки вероятностями не добьёшься. Чаще всего кривая ошибок выглядит так:



Чем выше она «выгибается» к верхнему левому углу, то есть чем ближе она к 1, тем лучше. А чем ближе она к линии `TPR=FPR` (от нижнего левого угла к верхнему правому), тем хуже. Это соответствует «случайной» модели: когда вы называете класс наугад.

Метрика roc-auc реализована в модуле `metrics` под названием `roc_auc_score`:

```
roc_auc = roc_auc_score(y_true, probabilities[:,1])
```

На вход этой функции подаются вектор реальных ответов `y_true` и вектор вероятности отнесения объекта к классу `"1"`. Для этого из вектора пар вероятностей отнесения объекта к классу `"0"` и `"1"`, берут лишь вторые числа (`probabilities[:,1]`). На выходе этой функции получаем одно число в пределах от 0 (чаще от 0.5) до 1.

Порог и баланс классов

Алгоритм логистической регрессии, подбирая наиболее оптимальные веса для функции `z(x)` функцией сигмоиды `f(z)`, вычисляет вероятность наступления события. В зависимости от этой вероятности вычисляется финальный ответ: `"0"` или `"1"`. Это происходит через сравнение с определённым порогом — неформально его называют **трешхолд** (англ. *threshold*, «порог»). Чаще всего он равен 0.5. Почему? На самом деле многие алгоритмы на выходе метода `predict_proba()` выдают не совсем вероятность отнесения к классу, а скорее степень уверенности модели в принадлежности к нему. Если модель уверена более чем на 0.5, что объект относится к классу `"1"`, тогда ответ `"1"`, иначе — `"0"`. Разберём, в каких случаях вероятность и «уверенность» можно считать одним и тем же, а когда — не совсем.

Представьте, что половина объектов в обучающей выборке относится к классу `"0"`, а половина — к `"1"`. Такие выборки называют **сбалансированные.**

Без алгоритмов и знаний о новом объекте невозможно точно сказать, к какому классу он принадлежит. С вероятностью 50% он может быть как класса `"0"`, так и `"1"`: изначальная базовая вероятность, что объект относится к каждому из классов, равна 0.5. Тогда эту вероятность можно считать степенью *уверенности* в том, относится объект к классу `"1"` или `"0"`. Если с учётом признаков, которые дают нам ценную дополнительную информацию об объекте, можно сказать, что вероятность отнесения к классу `"1"` больше 0.5 — значит всё-таки уверенность больше в классе `"1"`, чем `"0"`. Отсюда и такой порог: к `"1"` относим все объекты, для которых `predict_proba()` для класса `"1"` выдала значение больше, чем 0.5.

В учебных целях вы можете решать рафинированные — упрощённые и не всегда правдоподобные — задачи на бинарную классификацию, где объектов одного класса примерно столько же, сколько и объектов второго. Скажем, если бы в мешке было одинаковое число как белых, так

и чёрных шаров. На практике классы редко сбалансированы: один сильно преобладает над другим, и результат отработки `predict_proba()` — это скорректированная на баланс классов вероятность.

Именно на несбалансированных классах начинаются сложности с метриками, основанными на бинарных ответах, а результат `predict_proba()` уже нельзя интерпретировать как вероятность класса. Однако многие алгоритмы нормируют вероятности или применяют другие математические методы так, чтобы результат `predict_proba()` выражал именно степень уверенности модели в окончательном бинарном ответе. Для этой степени вы по-прежнему можете задать свой трешхолд при определении бинарных классов. И таким образом скорректировать наиболее важную для вас метрику.

Дерево принятия решений

Принципиально новый алгоритм поиска правильного ответа в задачах классификации и регрессии — **дерево принятия решений**. Если до этого вы предполагали, что ищете ответ — значение целевой функции — как сумму взвешенных значений признаков (линейной функции), то деревья выбирают ответ **сценарным подходом**. Последовательностью признаков алгоритм пытается так разбить выборку, чтобы группы в конце дерева (в листьях) были максимально отделимы на основании наблюдаемых данных.

Чем деревья хороши:

- Они легко интерпретируемы. В бизнесе это часто становится ключевым. Более сложные алгоритмы (о которых мы поговорим позже) напоминают «чёрный ящик» и неясно, по какому именно принципу они принимают решения.
- Годятся и для классификации, и для регрессии — в этом случае в листьях дерева будут числа, а не значение класса.
- Достаточно быстры.
- Не требуют серьёзной предобработки данных: деревья не чувствительны к масштабу признаков и устойчивы к их мультиколлинеарности.

Чем деревья плохи:

- Они сильно переобучаются. Напомним, это значит, что ваша модель слишком сильно подгонит параметры или разбиение веток под обучающую выборку. На *train* она будет работать очень хорошо, а вот на новых данных терпит фиаско. Иногда при большом количестве признаков вы можете строить ветку так, что в листе будет один объект, или совсем немного. По сути дерево подстроится под конкретную обучающую выборку, пытаясь оставить в листе объекты лишь одного класса.

Чтобы этого избежать, проводят процедуру «обрезки» дерева — **prunning.** Она заключается в том, чтобы в определённый момент обрезать ветку, не позволяя дереву переобучаться. Эффект переобучения особенно заметен на небольших выборках.

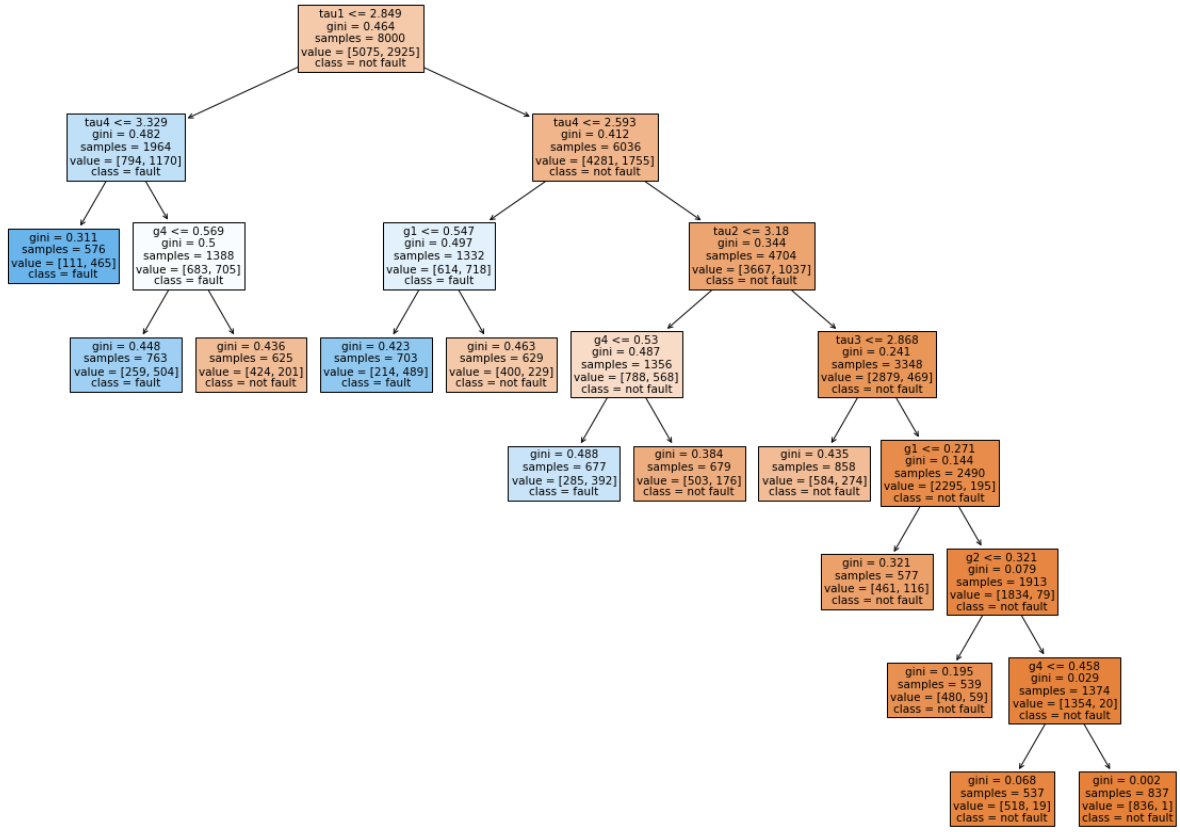
В `sklearn` есть модуль `tree`, где реализованы алгоритмы `DecisionTreeClassifier` и `DecisionTreeRegressor`:

```
tree_model = DecisionTreeClassifier()
tree_model.fit(X_train, y_train)
y_pred = tree_model.fit(X_test)
```

Бывает нужно визуализировать полученное дерево решений. В том же модуле есть функция **plot_tree()**:

```
plt.figure(figsize = (20,15)) # задайте размер фигуры, чтобы получить крупное изобр
ажение
plot_tree(tree_model, filled=True, feature_names = X_train.columns, class_names =
 ['not fault', 'fault'])
plt.show()
```

В результате получите что-то вроде:



Это дерево построено для модели `tree_model =DecisionTreeClassifier(min_samples_leaf=500)`, решающей задачу бинарной классификации (`"fault"`/`"not fault"`**).** Выборка порядка 10 000 объектов. В каждом узле указано условие, по которому идёт дальнейшее ветвление, а также наиболее вероятное значение ответа в соответствующей данному узлу выборке. Обратите внимание: при большой выборке лучше указать ограничение на минимальное число объектов в листе, например, `min_samples_leaf=500`. Если так не сделать, дерево будет строиться до последнего, станет слишком ветвистым и неудобным для отрисовки.

Ансамбли деревьев: случайный лес и градиентный бустинг

Ансамбли — серьёзные модели машинного обучения. Они мощные и позволяют отражать сложные зависимости между данными. Однако за «силу» ансамблей приходится платить их интерпретируемостью. Такие алгоритмы часто называют **«чёрными ящиками»**: по ним трудно понять,

какие же признаки повлияли на прогноз для конкретного наблюдения. Вам решать, чем именно вы готовы пожертвовать, решая задачи бизнеса.

Одно дерево хорошо, а много — лучше

Идея ансамбля моделей проста. Пусть вы решили задачу бинарной классификации моделью `m_1` на основе дерева принятия решений. Получили значение метрики `roc_auc_score_1 = 0.8`. Однако все модели склонны переобучаться. Один алгоритм, обученный на всех наблюдениях *train* и всех признаках, может сильно ошибаться на валидации. Есть идея взять ещё одну модель — `m_2`, компенсирующую ошибки первой. Сама по себе она также может давать не очень высокий `roc_auc`. Например, `roc_auc_score_2=0.79`. Можно поменять условия ещё раз и построить ещё одну модель `m_3` и так далее до модели `m_n`. В итоге каждая модель из `n` будет подвержена разным случайным факторам в зависимости от своих отличий, но при этом принесёт собственное сакральное знание о наблюдениях. Можно попытаться учесть их все одновременно — получить сводную модель `M`, которая будет давать результат лучше, чем каждая в отдельности — скажем, `roc_auc_score = 0.91`. Такую сводную модель будем называть **сильной** (говорят «сильным классификатором»), а модели, на основании которых она построена — **слабыми** (или «слабыми классификаторами»).

Два основных типа ансамблевых моделей: **случайный лес и градиентный бустинг**. В них в качестве слабых моделей чаще всего берут деревья принятия решений. Выбор деревьев не случаен. В отличие от, например, линейных моделей, у деревьев для каждого отдельного слабого классификатора можно менять не только подвыборки наблюдений и признаков, но и другие параметры. Например, ограничить глубину дерева или минимальное число объектов в каждом узле. Поэтому деревьями получают гораздо больше моделей для решения одной и той же задачи. Усреднение работает лучше, когда модели отличаются и компенсируют ошибки друг друга. А вот усреднение похожих моделей с одинаковыми недостатками не приведёт к такому эффекту.

Случайный лес (англ. *Random Forest*)

Первый тип ансамбля моделей — **случайный лес**. Он генерирует множество различных **независимых друг от друга деревьев** слегка разными способами (берёт разные подвыборки, разные признаки), а на основании их ответов формирует итоговое решение. Алгоритм случайного

леса **усредняет ответы** всех деревьев (в задаче регрессии) или **выбирает голосованием** (в классификации) тот ответ, который большинство деревьев в лесу считает правильным.

Говоря об ансамблях моделей, применяют термин **бэггинг** (англ. *bagging*). Он означает усреднение моделей, обученных на разных подвыборках. А сам подход, при котором оценивают различные характеристики или прогнозы на основании множества различных подвыборок из исходной, называют **бутстреп** (англ. *bootstrap*)**.**

При работе с *Random Forest* помните, что деревья обучаются «параллельно» — независимо друг от друга. Обучение каждого дерева не зависит от результатов других. Это концептуальное отличие случайного леса от градиентного бустинга.

Градиентный бустинг (англ. *gradient boosting*)

Фундаментально идея та же — собираем много простых моделей (деревьев), чтобы они компенсировали ошибки и случайности друг друга. Только в этом случае компенсация ошибок происходит не за счёт усреднения, а через **обучение**.

В градиентном бустинге деревья обучаются **последовательно**: каждое следующее дерево строится с учётом результатов предыдущего. Вы уже знаете принцип работы градиентного спуска: когда, двигаясь в нужном направлении, последовательно на каждом следующем шаге стараются минимизировать ошибку. Идея градиентного бустинга аналогична: на определённом шаге есть модель, и она делает прогноз с какой-нибудь ошибкой. Тогда строят вторую модель, которая будет прогнозировать не исходную величину, а ошибку первой модели, и с учётом этого корректировать финальный прогноз. Третья модель спрогнозирует ошибку второй и так дальше. В финале прогнозируем конкретное наблюдение на основании всей этой последовательности. Говорим, что первая модель дала такой ответ, но к нему добавляем корректировку из второй, потом — из третьей и так, пока не дойдём до последней. Так на каждом следующем шаге (каждой следующей моделью) улучшают прогноз предыдущего. Бустинг означает «улучшение». А градиентный бустинг — «последовательное улучшение».

Реализация ансамблей деревьев

Случайный лес в `sklearn` реализован в модуле `ensemble`. Из этого модуля чаще всего применяют алгоритмы `RandomForestClassifier()` и

`RandomForestRegressor()` для задач классификации и регрессии, соответственно.

Синтаксис объявления, обучения и прогнозирования на примере задачи регрессии:

```
from sklearn.ensemble import RandomForestRegressor

# зададим алгоритм для новой модели на основе алгоритма случайного леса
rf_model = RandomForestRegressor(n_estimators = 100)
rf_model.fit(X_train, y_train)
y_pred = rf_model.predict(X_val)
```

Тут при объявлении модели мы задали `n_estimators` — число деревьев, на основании которых будем строить лес. При этом другие параметры дерева — например, глубину дерева `max_depth`, размер подвыборки признаков `max_features`, минимальное количество объектов в узле `min_samples_leaf` — мы не задавали и оставили их по умолчанию. На практике вы можете экспериментировать с этими значениями и смотреть, как это влияет на результат.

Градиентный бустинг в `sklearn` реализован в модуле `ensemble`. В нём алгоритмы `GradientBoostingClassifier()` и `GradientBoostingRegressor()` для задач классификации и регрессии, соответственно.

Синтаксис на примере задачи регрессии:

```
from sklearn.ensemble import GradientBoostingRegressor

# зададим алгоритм для новой модели на основе алгоритма градиентного бустинга
gb_model = GradientBoostingRegressor(n_estimators = 100)
gb_model.fit(X_train, y_train)
y_pred = gb_model.predict(X_val)
```

Вы также могли слышать об `xgboost`. Этот алгоритм из отдельной библиотеки часто применяют в соревнованиях по машинному обучению. Вы можете прочитать о нём тут:

https://xgboost.readthedocs.io/en/latest/

https://www.datacamp.com/community/tutorials/xgboost-in-python

Алгоритмы обучения без учителя: кластеризация

Задачи обучения без учителя сводятся к вычислению схожести между объектами. Когда модель её обнаружила, можно группировать сами объекты — решать **задачу кластеризации**. Можно вычислять и схожесть признаков — это потребуется для решения **задачи снижения размерности**.

Подробно разберём задачу кластеризации — она чаще всего встречается в бизнесе. Например, кластеризацию применяют:

- Для сегментации клиентов
- Для сегментации продуктов
- Для тематического моделирования

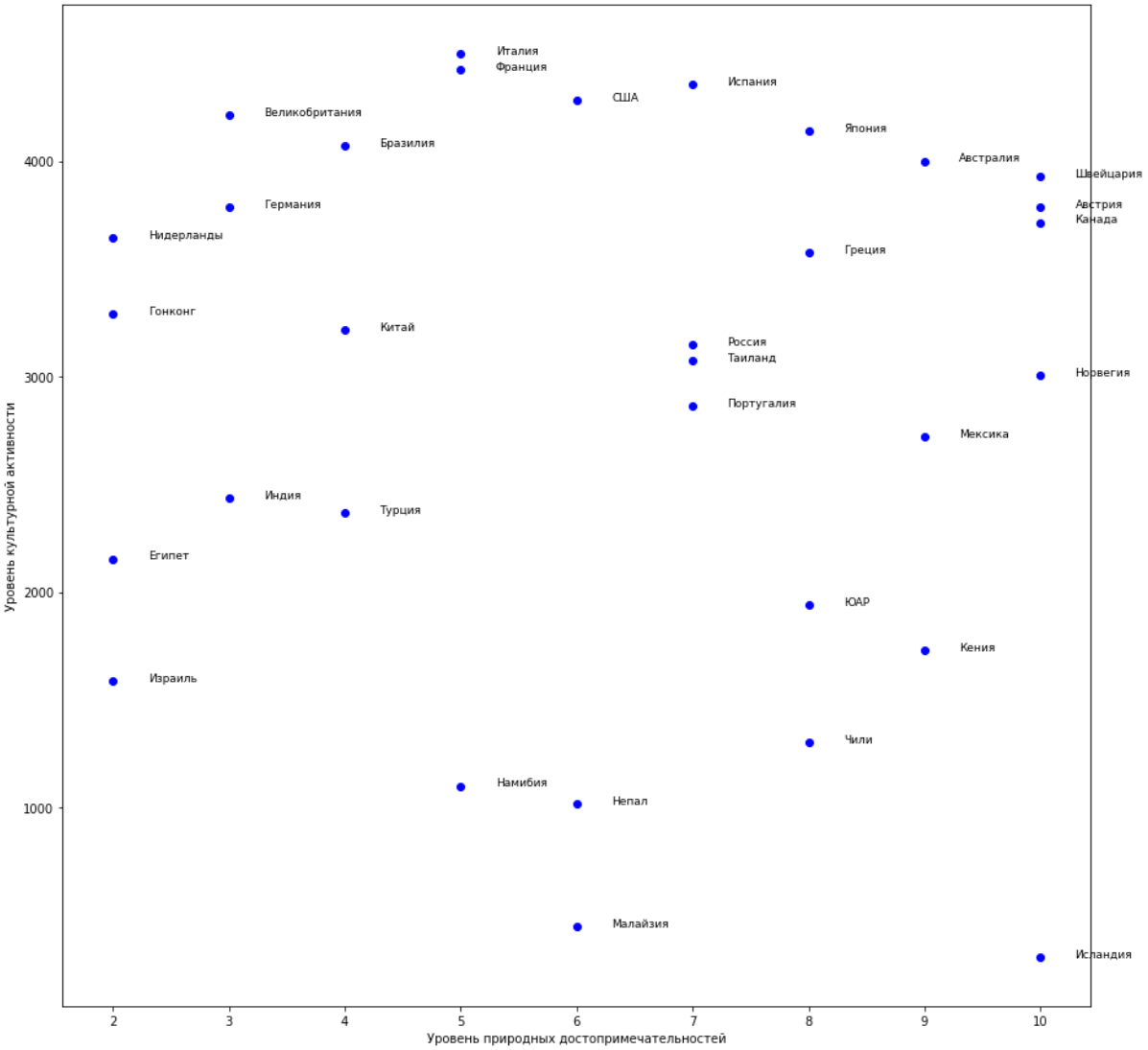
Причём здесь расстояние?

У схожих объектов и признаки близки, иначе говоря, находятся на небольшом *расстоянии* друг от друга. Интуитивно от термина «похожесть» перейдём к более привычному *расстоянию*.

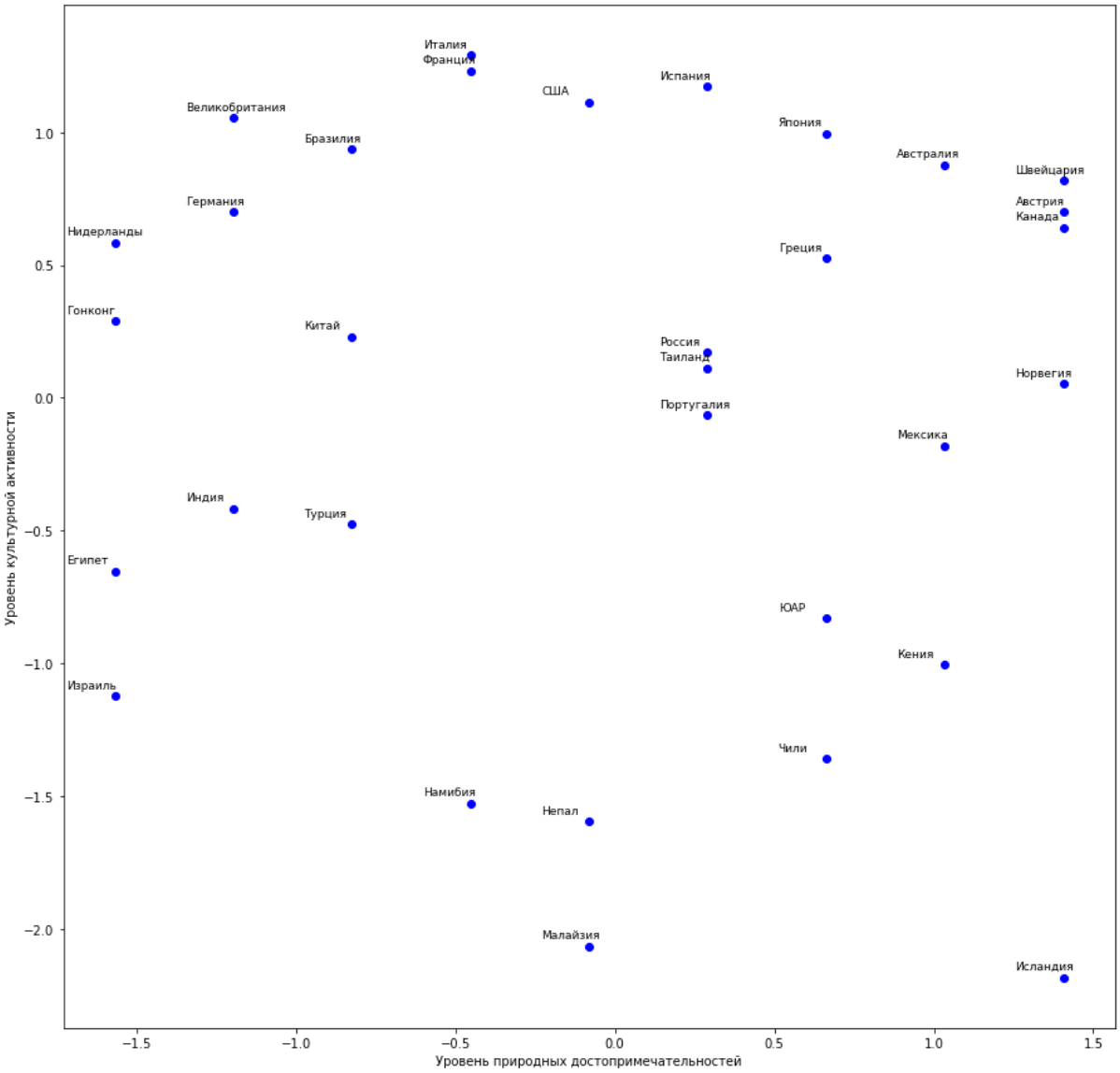
Отсюда, первый шаг задачи кластеризации: задать расстояние между объектами — получить численно выраженную степень их близости. А уже на основе рассчитанной близости группировать сами объекты. Определим функцию расстояния между объектами так:

$$\rho(x_1, x_2)$$

Пусть у каждой страны только 2 признака. Как задать расстояние между двумя объектами на основе данных по двум признакам? Можно нарисовать график, где по оси X откладывать один признак, а по оси Y — второй. Например, так:



Однако сперва хорошо бы привести признаки к единому масштабу знакомыми методами **нормализации** и **стандартизации**. Напомним, нормализация присвоит признакам значения в интервале от 0 до 1. Стандартизация приведёт значения признаков к виду стандартного нормального распределения: значения будут центрированы относительно 0, а средний разброс данных составит 1. После стандартизации, визуализация признаков может выглядеть так:



Зная значения признаков для любых двух объектов, как найти расстояние? Применим стандартную формулу расстояния (его ещё называют евклидовым):

$$\rho(x_1, x_2) = \sqrt{\sum_{i=1}^{n} (x_{1_i} - x_{2_i})^2}$$

Здесь `n` — число признаков. Тогда для случая, когда признака всего два, расстояние между двумя странами будет задаваться так:

$$\rho(x_1, x_2) = \sqrt{(x_{1_1} - x_{2_1})^2 + (x_{1_2} - x_{2_2})^2}$$

K-Means и агломеративная иерархическая кластеризация

Самые популярные алгоритмы кластеризации:

- **K-Means**
- **Агломеративная иерархическая кластеризация**

Первый метод быстрый и понятный, однако чувствительный к заданному числу кластеров. Второй посложнее, зато не требует заданного заранее числа кластеров и позволяет визуализировать соотношения между объектами.

K-Means (англ. «K средних»)

K-Means группирует объекты пошагово. Алгоритм основан на предположении, что число кластеров (групп) заранее известно. Это довольно сильное допущение, и часто выбор оптимального количества кластеров заслуживает решения отдельных задач.

Вот принцип работы пошагового алгоритма K-means:

1) Есть K кластеров. Алгоритм пошагово подбирает их центры и относит объекты к тому кластеру, чей центр ближе.

2) Центры корректируются (меняют своё положение) до тех пор, пока это позволяет минимизировать среднее расстояние от объектов каждого кластера до его центра.

3) Когда расстояние от объектов до центра перестаёт снижаться или сокращается несущественно, алгоритм останавливается и фиксирует разбиение, считая его оптимальным.

Разобравшись в устройстве алгоритма, запустим его в Python.

Алгоритм под названием `KMeans` реализован в модуле `sklearn.cluster`. Вот его синтаксис:

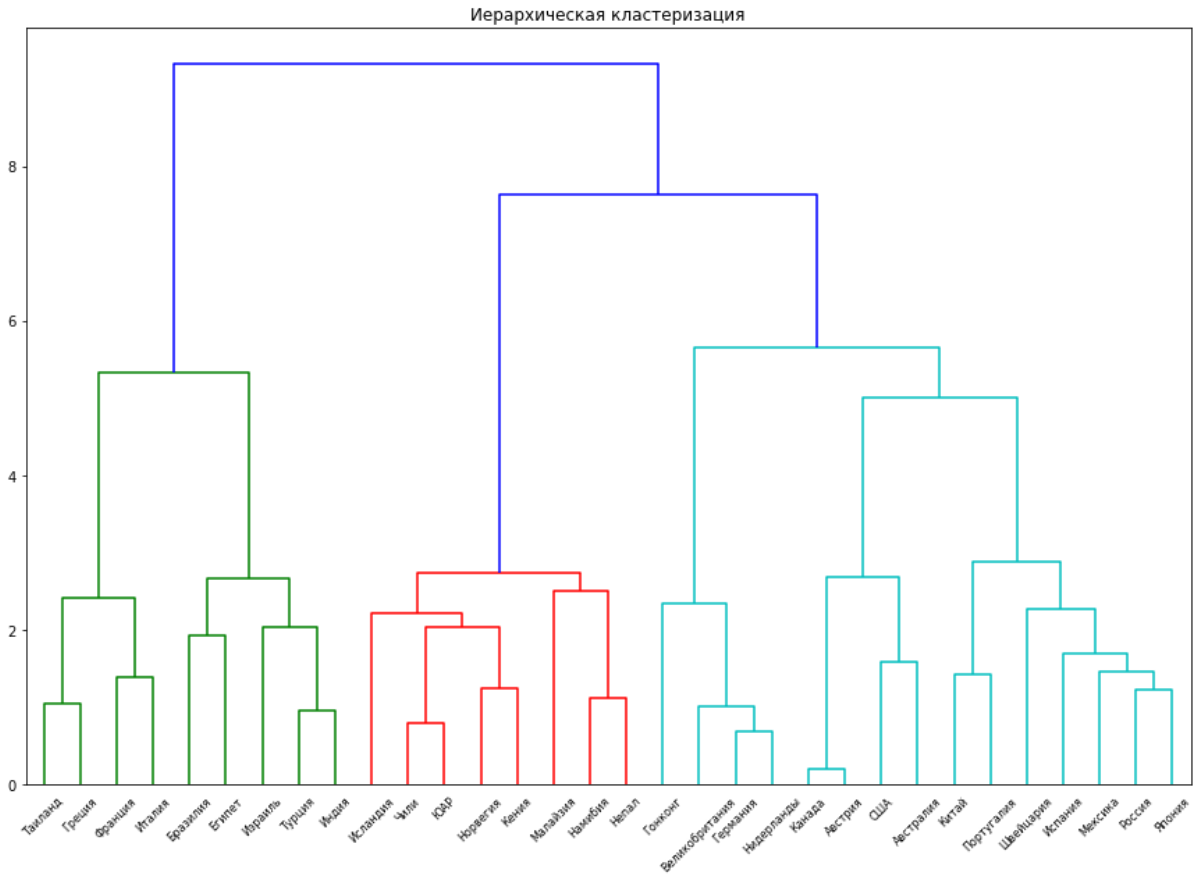
```
from sklearn.cluster import KMeans

# обязательная стандартизация данных перед работой с алгоритмами
sc = StandardScaler()
X_sc = sc.fit_transform(X)

km = KMeans(n_clusters = 5) # задаём число кластеров, равное 5
labels = km.fit_predict(X_sc) # применяем алгоритм к данным и формируем вектор клас
теров
```

В переменной `labels` сохраняются индексы предложенных алгоритмом групп. Алгоритм случайно назначает номер определённой группе, поэтому искать в этой цифре какой-то смысл не нужно: группа с индексом `2` не «ближе» к группе с индексом `3`, чем группа `1`. Важно, что объекты, которым модель присвоила один и тот же индекс, относятся к одному кластеру.

Агломеративная иерархическая кластеризация

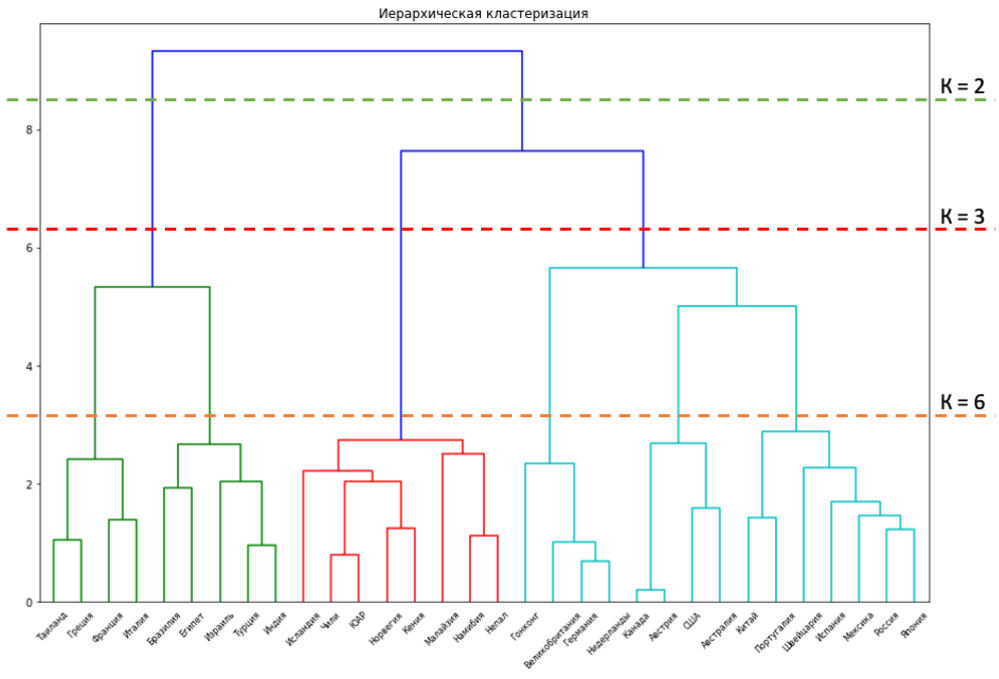
Суть иерархической кластеризации проста. Когда мы задали функцию расстояния, можно вычислить матрицу расстояний между всеми объектами, в ячейках которой будет попарное расстояние между двумя объектами. А ещё учтены все признаки объектов, а не только два.

На основе матрицы объект-признак можно последовательно объединять близкие кластеры. По ней оценить близость отдельных объектов, быстро найти самые близкие или самые удалённые друг от друга на глаз сложно. Однако расстояния между объектами и сама агломеративная иерархическая кластеризация хорошо визуализируются на специальных графиках — **дендрограммах:**



По вертикали расстояние между кластерами, а по горизонтали — объекты. Каждая горизонтальная «связка» соответствует расстоянию между объединяемыми объектами. Если двигаться по этому графику снизу вверх, сначала идут связки, которые соединяют отдельные объекты; потом связки, которые соединяют объекты с группами или одни группы объектов с другими. Если этот процесс не останавливать специально, он прекратится, когда останется только один гигантский кластер. Именно так и работает сама **иерархическая агломеративная** кластеризация. Этот способ кластеризации интуитивно понятный и наглядный. На каждом шаге алгоритм наращивает кластеры, присоединяя соседние. Потому он и называется агломеративным, ведь превращает кластеры в **агломерации** — так называют объединение соседствующих населённых пунктов. Важно понять — в какой именно момент процесс объединения следует остановить.

По дендрограмме можно визуально оценить, сколько кластеров должно быть. А также прикинуть расстояние, после которого мы перестаём объединять объекты. Вот пример того, как для большого количества объектов можно подбирать разные пороги расстояний:



Значение K — это число линий, пересечённых пунктирной линией.

Нарисуем дендрограмму в Python. Сперва из модуля для иерархической кластеризации `hierarchy` импортируем классы модели кластеризации

`linkage()` и `dendrogram()`:

```
from scipy.cluster.hierarchy import dendrogram, linkage
```

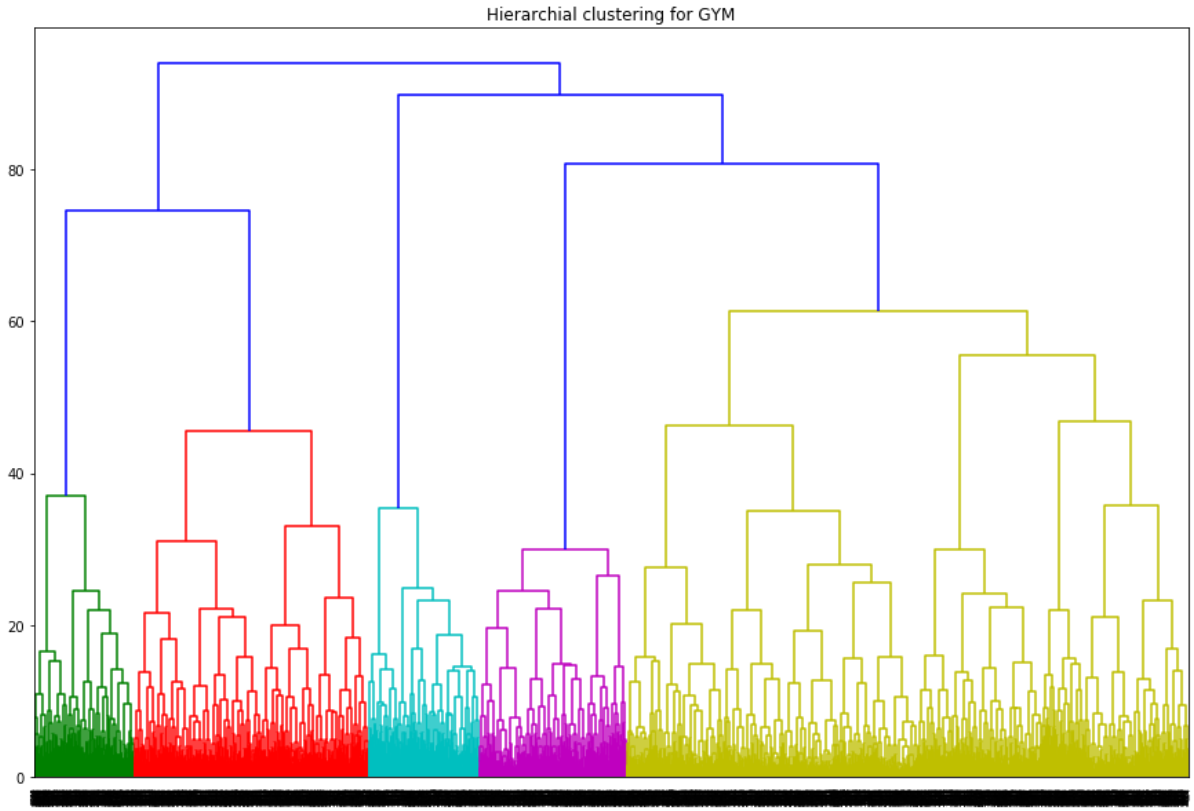
После этого выполним стандартизацию и передадим получившуюся стандартизированную таблицу в качестве параметра функции `linkage()`. Чтобы диаграмма получилась показательной, лучше передать параметру `method` значение `'ward'`:

```
# обязательная стандартизация данных перед работой с алгоритмами
sc = StandardScaler()
X_sc = sc.fit_transform(X)

linked = linkage(X_sc, method = 'ward')
```

В переменной linked сохранена таблица «связок» между объектами. Её можно визуализировать как дендрограмму:

```
plt.figure(figsize=(15, 10))
dendrogram(linked, orientation='top')
plt.title('Hierarchial clustering for GYM')
plt.show()
```

В результате получаем график:



Предложенное оптимальное число кластеров 5 — пять разных цветов на графике.

Сложность агломеративной кластеризации заключается не в устройстве самого алгоритма, а в вычислениях, которые машина совершает для построения дендрограммы. Расчёты попарных расстояний могут занять очень много времени. Потому при решении задачи кластеризации полезно строить дендрограмму на случайной подвыборке, а после оценки оптимального числа кластеров запустить более быстрый алгоритм K-Means.

Метрики для задач обучения без учителя

Забавный факт: существует **теорема невозможности Клейнберга**, которая гласит, что *оптимального алгоритма кластеризации не существует*. Это значит, что в силу отсутствия разметки, нет единственно верного способа оценить результат работы алгоритма. Все подходы к определению того, насколько хорошо разделили объекты, основаны на привлечении экспертов или эвристических оценках. Тем не менее, в бизнесе задачи

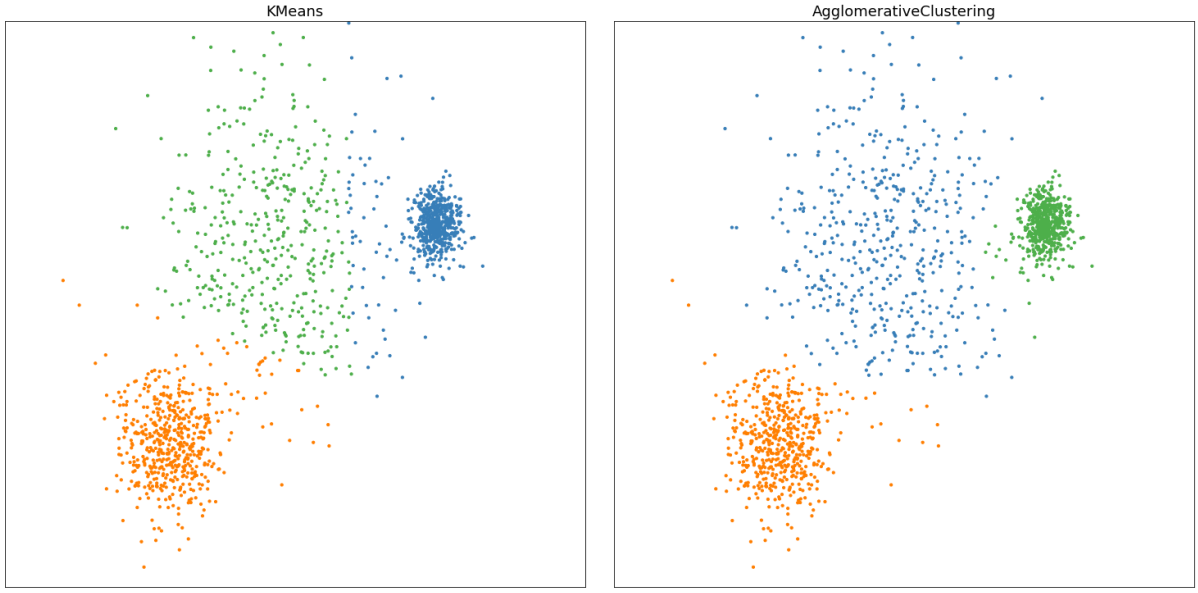
кластеризации очень востребованы. И оценивать качество их решения как-то нужно.

Действительно, в случае метрик для задач с размеченными данными всё было просто: вот прогнозы, вот реальные значения целевой переменной, а уж специалисты по машинному обучению разными способами показывают, насколько и как именно они различаются.

Но как оценить, насколько правдива наша модель, если ответов нет? Как вообще оценивать качество алгоритмов, которые группируют объекты на основании «похожести»?

Ответ кроется в самом способе постановки задачи. Раз за основу «похожести» берём расстояние между объектами, значит можно рассчитать такие основанные на понятии расстояния метрики, чтобы оценить, насколько хорошо модели делят объекты.

Вы поделили клиентов, применив K-Means и иерархическую кластеризацию. Как понять, какой алгоритм сгруппировал лучше? Допустим, у вас всего 2 признака и итоговая кластеризация выглядит так:



Скорее всего, вы будете смотреть, насколько «разделимы» между собой кластеры. Есть ли визуальные отличия на рисунке между объектами разных кластеров?

Чтобы формализовать это решение, перейдём к понятиям **внутрикластерных** и **межкластерных расстояний**. Несложно догадаться, что внутрикластерное — расстояние между объектами одного кластера, а межкластерное — между объектами разных кластеров. Кластеризация

тем лучше, чем сильнее отличаются внутрикластерные и межкластерные расстояния.

Рассмотрим популярную **метрику силуэта** (англ. *silhouette score*). Другие метрики на неё похожи. Метрика силуэта показывает, насколько объект своего кластера похож на свой кластер больше, чем на чужой. Вот её формула:

$$Silhouette = \frac{1}{n} * \sum_{c_k \in C} \sum_{x_i \in c_k} \frac{b(x_i, c_k) - a(x_i, c_k)}{max[a(x_i, c_k), b(x_i, c_k)]}$$

где `n` — количество наблюдений;

`c_k` — конкретный кластер;

`x_i` — конкретный объект кластера `c_k`;

`a(x_i, c_k)` — среднее расстояние от объекта `x_i` до других объектов кластера. Характеризует **компактность** (кучность) кластера;

`b(x_i, c_k)` — среднее расстояние от объекта `x_i` до объектов другого кластера. Характеризует **отделимость** кластера от других.

Синтаксис метода:

```
from sklearn.metrics import silhouette_score

silhouette_score(x_sc, labels)
```

На вход передаём нормализованную или стандартизованную матрицу признаков и метки, которые спрогнозировал алгоритм кластеризации, в виде списка. Значение метрики силуэта принимает значения от -1 до 1. Чем ближе к 1, тем качественнее кластеризация.