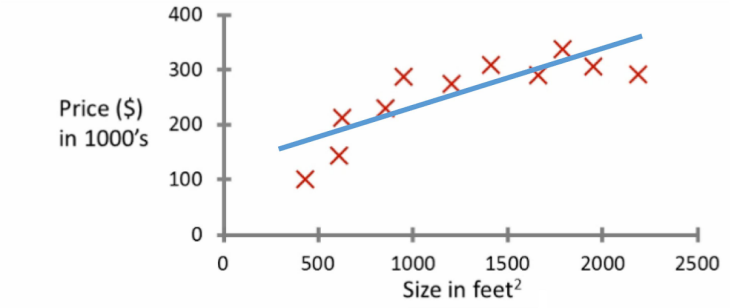
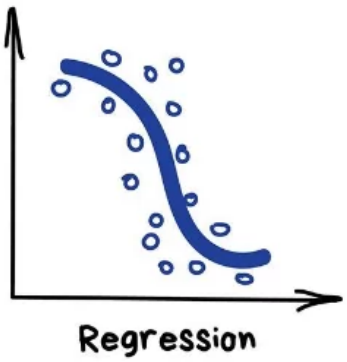
Линейные модели — это модели, отображающие зависимость целевого признака от факторов в виде линейной взаимосвязи.

Линейные модели



Регрессия



Регрессия — это класс задач обучения с учителем, когда по определённому набору признаков объекта необходимо предсказать **числовую целевую переменную**.

Цель обучения — построить модель, которая бы отражала зависимость между признаками и целевой числовой переменной.

Когда зависимость принимается линейной, такая модель называется **линейной регрессией**.

Линейная регрессия

Линейная регрессия (Linear Regression) — одна из простейших моделей для решения задачи регрессии. Главная гипотеза состоит в том, что рассматриваемая зависимость является линейной.

Общий вид модели в случае, когда целевая переменная зависит от m факторов, будет иметь следующий вид:

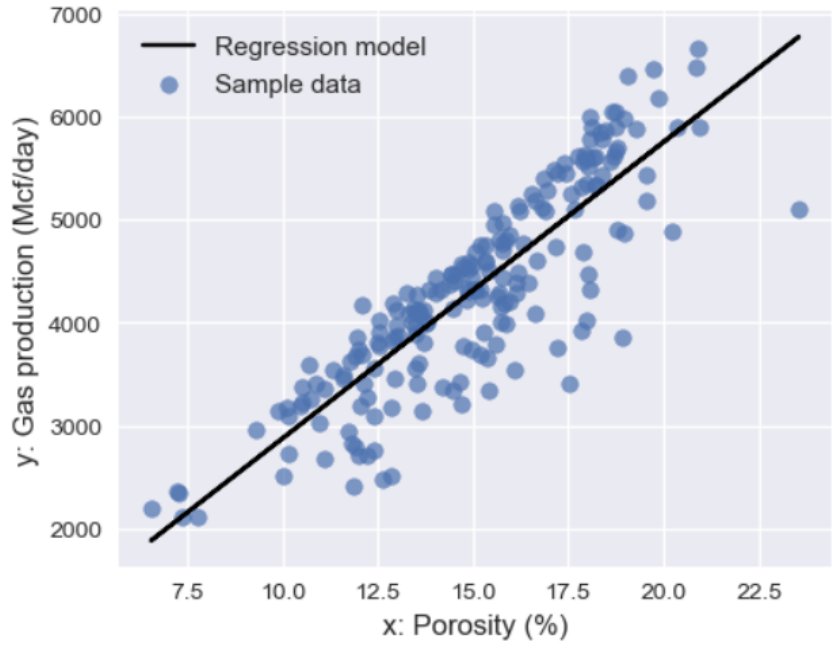
$$\hat{y} = \omega_0 + \omega_1 x_1 + \omega_2 x_2 + ... + \omega_m x_m$$

Геометрическая интерпретация. 2D-случай

В случае, когда целевой признак y зависит от одного только фактора x, уравнение модели линейной регрессии — это уравнение прямой в 2D-пространстве:

$$\hat{y} = \omega_0 + \omega_1 x$$

Пример:

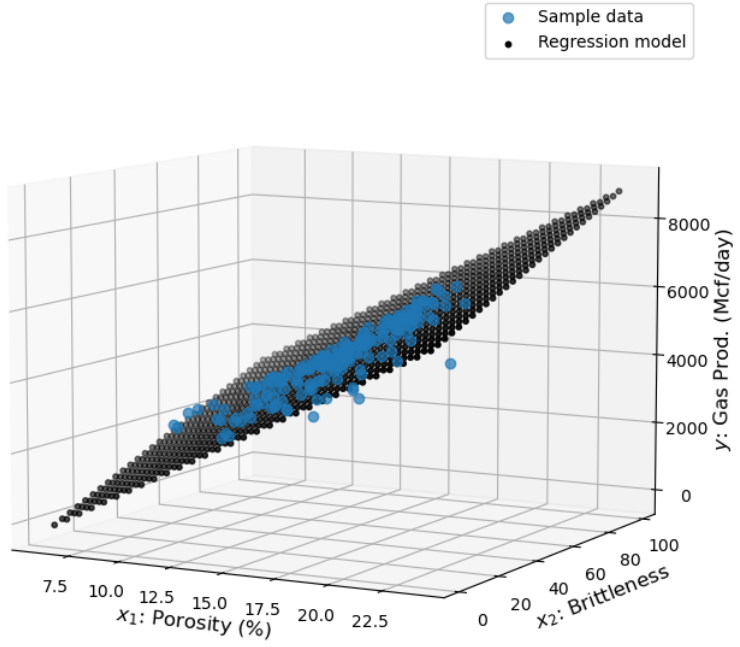


Геометрическая интерпретация. 3D-случай

Когда целевой признак y зависит от двух факторов x_1 и x_2, уравнение модели линейной регрессии — это уравнение плоскости в 3D-пространстве:

$$\hat{y} = \omega_0 + \omega_1 x_1 + \omega_2 x_2$$

Пример:



Геометрическая интерпретация. Общий случай

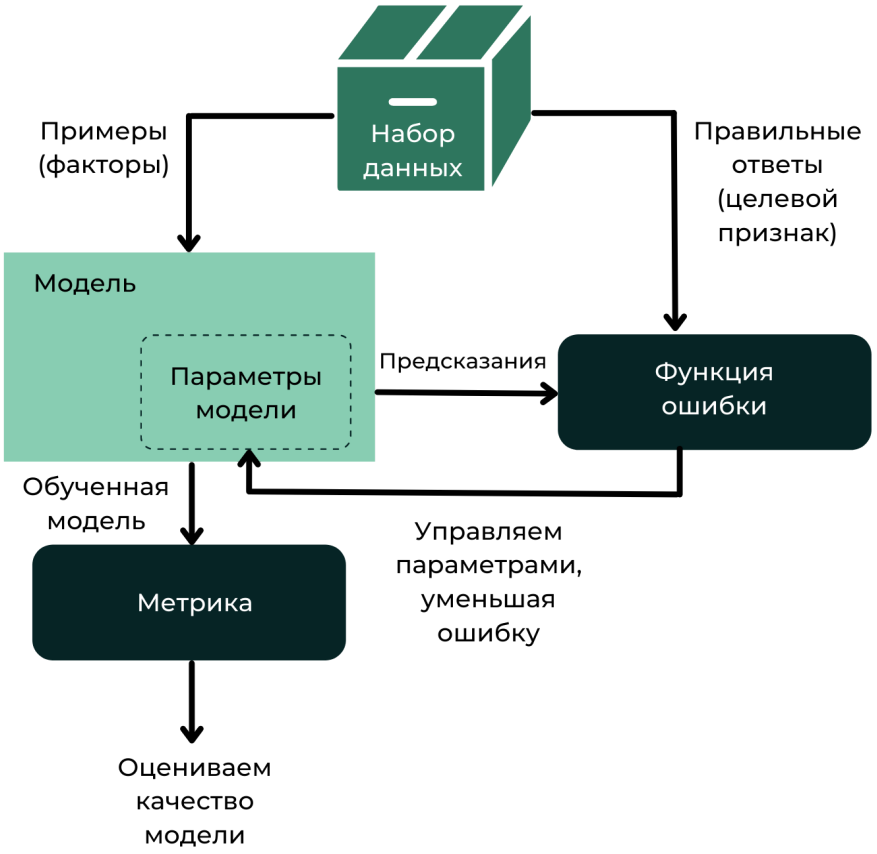
Пусть у нас есть m факторов $\{x_1, x_2, ..., x_m\}$, от которых зависит целевая переменная y.

$$\hat{y} = \omega_0 + \omega_1 x_1 + \omega_2 x_2 + ... + \omega_m x_m = \omega_0 + \sum_{j=1}^{m} \omega_j x_j$$

В геометрическом смысле данное уравнение описывает плоскость в $(m + 1)$-мерном пространстве (m факторов + один целевой признак отложены по осям координат). Такую плоскость называют **гиперплоскостью**.

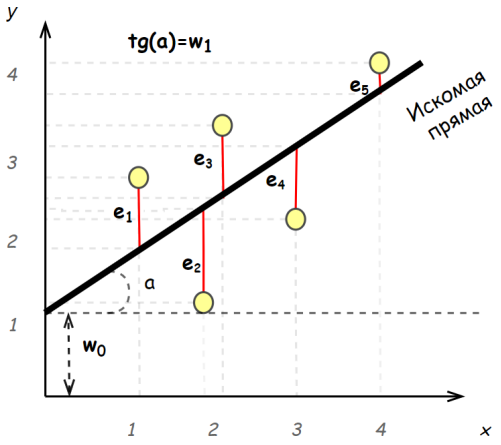
Поиск параметров линейной регрессии

Поиск параметров модели производится по принципу минимизации эмпирического риска (минимизации функции потерь):



Функция ошибки в линейной регрессии — **средний квадрат разности ошибки (MSE)**.

Составление функции ошибки:



Рассчитать ошибки e_i (на рисунке они отмечены красными отрезками):

$e_i = |y_i - \hat{y}_i|$,

где $\hat{y}_i$ — это результат подстановки i-ого значения x в модель линейной регрессии.

Возвести все ошибки в квадрат и вычислить среднее:

$$MSE = \frac{\sum_{i=1}^{n} e_i^2}{n} = \frac{\sum_{i=1}^{n} (y_i - \hat{y}_i)^2}{n} = \frac{\sum_{i=1}^{n} (y_i - \omega_0 - \omega_1 x_i)^2}{n}$$

Это и будет функция ошибки, которую мы будем минимизировать, управляя параметрами w_0 и w_1:

$$MSE = \frac{\sum_{i=1}^{n} (y_i - \hat{y}_i)^2}{n} = \frac{\sum_{i=1}^{n} (y_i - \omega_0 - \omega_1 x_i)^2}{n} \to min_{\omega}$$

В общем случае, когда X — это таблица из n наблюдений и m признаков, постановка задачи оптимизации MSE выглядит следующим образом:

$$MSE = \frac{\sum_{i=1}^{n} (y_i - \hat{y}_i)^2}{n} = \frac{\sum_{i=1}^{n} (y_i - \omega_0 - \sum_{j=1}^{m} \omega_j x_{ij})^2}{n} \to min_{\omega},$$

где x_{ij} — значение, которое находится в i-ой строке и j-ом столбце таблицы наблюдений.

Аналитическое решение. Метод наименьших квадратов

> Метод поиска параметров линейной регрессии называется **методом наименьших квадратов** (сокращённо — **МНК**). В англоязычной литературе часто можно встретить аббревиатуру **OLS** (**Ordinary Least Squares**).

Аналитическое решение по МНК для общего случая:

$\hat{y} = \omega_0 + \omega_1 x_1 + \omega_2 x_2 + ... + \omega_m x_m = \overline{\omega} \cdot \overline{x}$

$\overline{\omega} = (\omega_0, \omega_1, \omega_2, \ldots, \omega_m)$ — вектор параметров
$\overline{x} = (1, x_1, x_2, \ldots, x_m)$ — вектор признаков

$$\overline{\omega} = (X^T X)^{-1} X^T y = Q X^T y$$

Данная матричная формула позволяет найти неизвестные параметры линейной регрессии в виде вектора $\overline{\omega} = (\omega_0, \omega_1, \omega_2, \ldots, \omega_m)$. Найденные коэффициенты называют **решением задачи линейной регрессии**.

Аналитическое решение с помощью NumPy

```
def linear_regression(X, y):
    #Создаём вектор из единиц
    ones = np.ones(X.shape[0])
    #Добавляем вектор к таблице первым столбцом
    X = np.column_stack([ones, X])
    #Вычисляем обратную матрицу Q
    Q = np.linalg.inv(X.T @ X)
    #Вычисляем вектор коэффициентов
    w = Q @ X.T @ y
    return w
```

Аналитическое решение с помощью sklearn

Обучение (поиск параметров) — метод `fit()`:

```
#Создаём объект класса LinearRegression
lr = linear_model.LinearRegression()
#Обучаем модель — ищем параметры по МНК
lr .fit(X, y)
```

Предсказание — метод `predict()`:

```
y_predict = lr.predict(X)
```

Посмотреть коэффициенты — атрибуты `coef_` и `intercept_`:

`lr .coef_` — коэффициенты при факторах
`lr.intercept _` — свободный член

Составление таблицы из факторов и коэффициентов при них:

```
#Составляем таблицу из признаков и их коэффициентов
w_df = pd.DataFrame({'Features': features, 'Coefficients': lr_full .coef_})
#Составляем строчку таблицы со свободным членом
intercept_df =pd.DataFrame({'Features': ['INTERCEPT'], 'Coefficients': lr_full
.intercept_})
coef_df = pd.concat([w_df, intercept_df], ignore_index=True)
```

Каждый из коэффициентов в модели показывает, на сколько в среднем изменится целевой признак при увеличении параметра на единицу.

Пример визуализации ошибок:

```
#Визуализируем ошибки
fig, ax = plt.subplots(figsize=(12, 6)) #фигура + координатная плоскость

#Ошибки модели на всех факторах
y_errors = y - lr.predict(X)
#Для удобства визуализации составим DataFrame из ошибок
errors_df = pd.DataFrame(
    {'Errors': y_errors}
)
#Строим boxplot для ошибок
sns.boxplot(data=errors_df, orient='h', ax=ax)
ax.set_xlabel('Model errors') #название оси абсцисс
```

Требования к данным при подаче в модель линейной регрессии:

- ➔ отсутствие пропущенных значений,
- ➔ отсутствие выбросов,
- ➔ кодированные категориальные признаки.

Метрики регрессии

> **Метрика** — это численное выражение качества моделирования.

Средняя абсолютная ошибка — MAE (Mean Absolute Error)

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

Средняя абсолютная ошибка в процентах — MAPE (Mean Absolute Percent Error)

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

Средняя квадратическая ошибка — MSE (Mean Squared Error)

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

Корень из средней квадратической ошибки — RMSE (Root Mean Squared Error)

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

Коэффициент детерминации (R^2)

$R^2 = 1 - \frac{MSE}{MSE_{mean}}$, где

$$MSE_{mean} = \frac{\sum_{i=1}^{n} (y_i - \bar{y})^2}{n}$$

Название	Формула	Интерпретация и применение	Достоинства	Недостатки	Функция в модуле metrics библиотеки sklearn
MAE	$\frac{\sum_{i=1}^{n} \lvert y_i - \hat{y}_i \rvert}{n}$	Помогает оценить абсолютную ошибку: насколько в среднем число в предсказании разошлось с реальным числом.	Удобно интерпретировать. Измеряется в тех же единицах, что и целевой признак. Несильно искажается при наличии выбросов.	Не поможет, если необходимо сравнить модели, предсказывающие одно и то же по разным признакам.	`mean_absolute_error()`
MAPE	$\sum_{i=1}^{n} \frac{\lvert y_i - \hat{y}_i \rvert}{\lvert y_i \rvert} \frac{100\%}{n}$	Помогает абстрагироваться от конкретных чисел и оценить абсолютную ошибку в процентах.	Легко интерпретировать. Используется в задачах, где неизвестно, какое значение метрики считать приемлемым.	Плохо подходит для задач, в которых важны конкретные единицы измерений. Лучше использовать в паре с MAE, чтобы знать абсолютную ошибку и её значение в процентах.	`mean_absolute_percentage_error()`
MSE	$\frac{\sum_{i=1}^{n} (y_i - \hat{y}_i)^2}{n}$	Интерпретации нет. Используется в задачах, где критически важны большие ошибки, например при предсказании	Каждая ошибка вносит свой квадратичный штраф, большие расхождения между предсказан	Измеряется в квадратах единиц, поэтому менее доступна для понимания. Искажается	`mean_square_error()`

		координат полёта.	ием и истиной увеличивают штраф.	при наличии выбросов. Не поможет, если нужно сравнить модели, предсказывающие одно и то же по разным признакам.	
RMSE	$\sqrt{\frac{\sum_{i=1}^{n}(y_i-\hat{y}_i)^2}{n}}$	Можно трактовать как стандартное отклонение предсказаний от истинных ответов. Используется в тех же задачах, что и MSE.	Имеет те же преимущества, что и MSE, но более удобна для понимания (измеряется в тех же единицах, что и целевая переменная).	Не поможет, если нужно сравнить модели, предсказывающие одно и то же по разным признакам.	Отдельная функция отсутствует, но можно извлечь корень из результата функции `mean_square_error()`.
R^2	$1 - \frac{MSE}{MSE_{mean}}$	Помогает понять, какую долю разнообразия (дисперсии) смогла уловить модель в данных. Позволяет сравнить, насколько модель лучше, чем простое предсказание средним.	Можно сравнивать модели, обученные на разных признаках. Легко оценить качество модели: измеряется от -∞ до 1. Удовлетворительным показателем считается показатель выше 0.5.	Чувствительна к добавлению новых данных. Чувствительна к выбросам, так как основана на MSE.	`r2_score()`

Расчёт метрик на Python

```
from sklearn import metrics
```

- `mean_absolute_error()` — расчёт MAE;
- `mean_square_error()` — расчёт MSE;
- `mean_absolute_percentage_error()` — расчёт MAPE;
- `r2_score()` — расчёт коэффициента детерминации R^2.

Примечание. Для расчёта метрики RMSE нет специальной функции, однако мы знаем, что для её расчёта достаточно извлечь квадратный корень из MSE.

По причине реализации `mean_absolute_percent_error()` функция возвращает результат НЕ в процентах, а в долях. Чтобы отобразить результат в процентах, необходимо умножить его на 100.

Недостатки аналитического решения

$$\overline{\omega} = (X^T X)^{-1} X^T y = Q X^T y$$

1. Кубическая сложность вычисления обратной матрицы. $Q = (X^T X)^{-1}$
2. Невозможность инкрементального обучения.
3. Матрица $Q = (X^T X)^{-1}$ может не существовать.

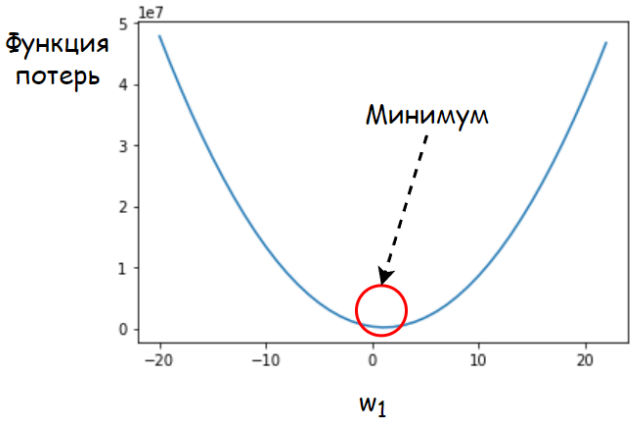
Численное решение. Градиентный спуск

Искать минимум функции потерь можно не только аналитическим, но и численным способом.

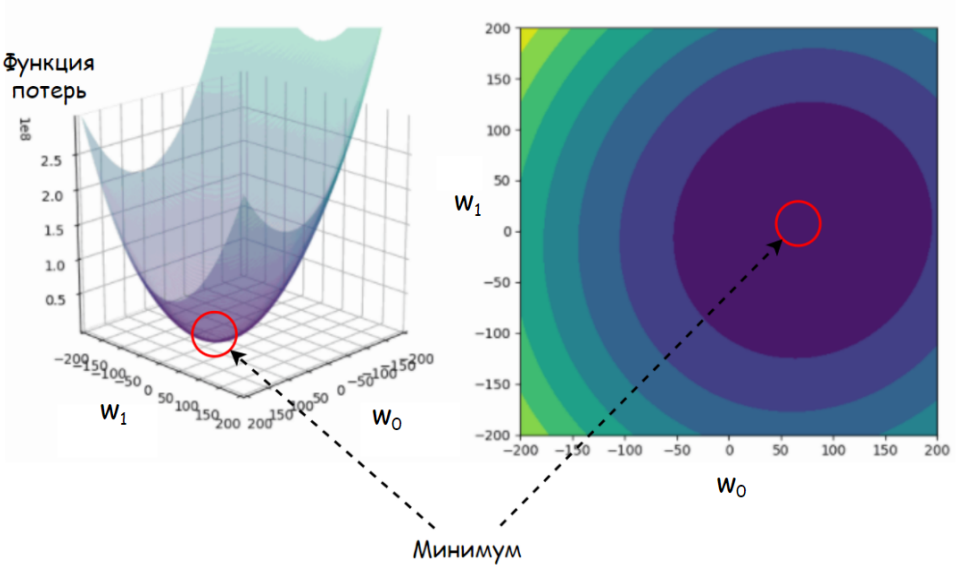
Градиентный спуск (Gradient descent) — самый используемый **алгоритм** минимизации функции потерь. Он применяется почти в каждой модели машинного обучения и является наиболее простым в реализации из всех методов численной оптимизации.

Пространство, в котором находится функция потерь — это **пространство параметров** ω нашей модели. То есть это система координат, в которой по осям отложены все возможные значения параметров ω.

Случай одного параметра:



Случай двух параметров:



1. **Проинициализировать значения параметров.**

   Выбрать начальную точку в пространстве, из которой мы будем двигаться.

2. **Повторять до тех пор, пока длина градиента не приблизится к 0.**

   2.1. Вычислить градиент функции потерь $\nabla L(\omega)$.

Это будет означать найти направление и вектор скорости роста функции потерь.

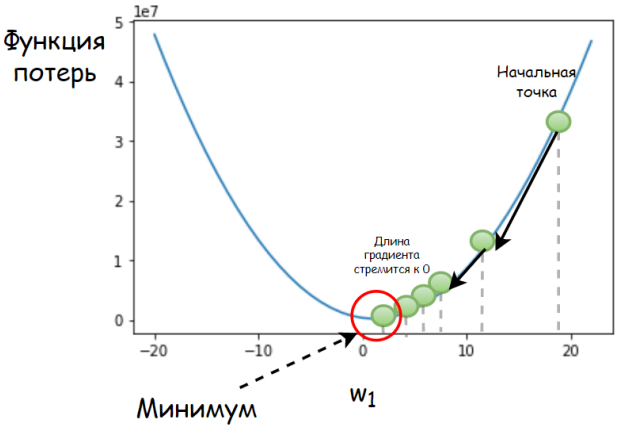
2.2. Обновить параметры модели, сдвинув их в сторону антиградиента.

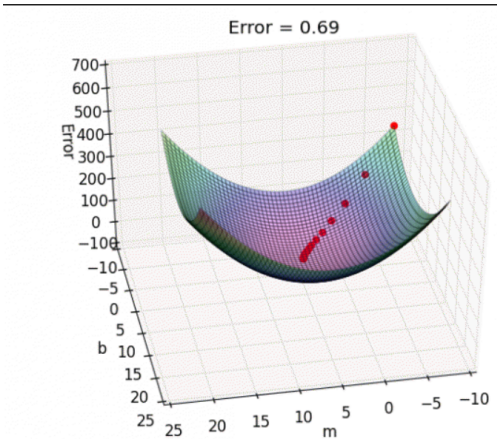
Из текущей точки нужно перейти в новую точку, в сторону убывания высоты ландшафта.

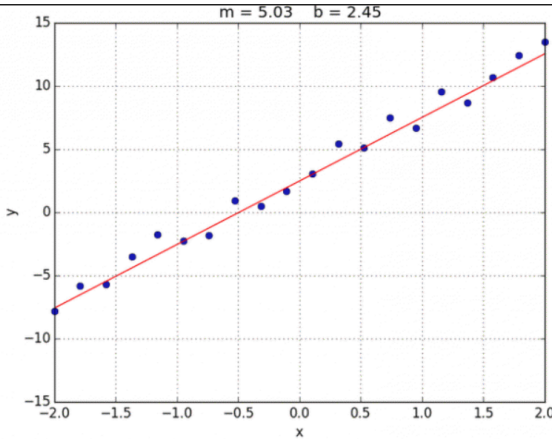
Для обновления координат точки используем формулу:

$$\omega^{(k+1)} = \omega^{(k)} - \eta \nabla L(\omega^{(k)})$$

Примеры пошаговой работы градиентного спуска:







Сходимость зависит от многих факторов, главные из которых:

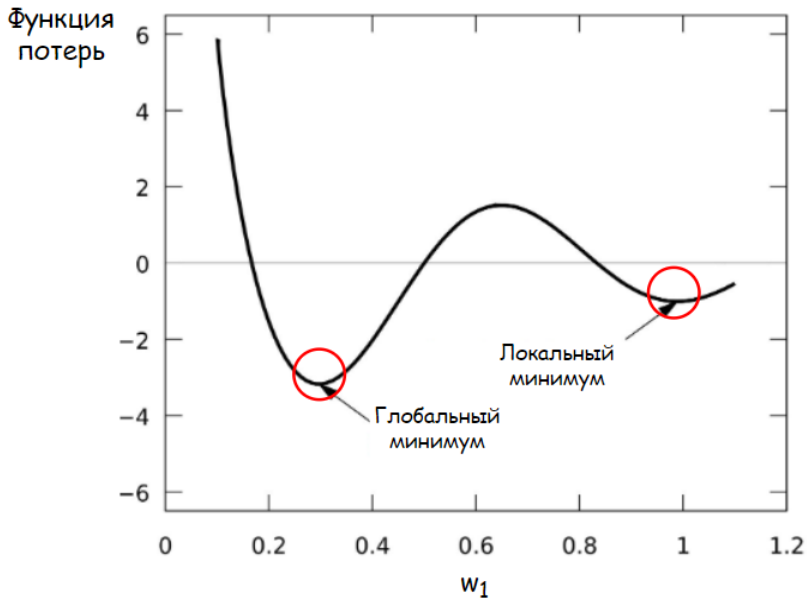
- сложности зависимости и сложности функции потерь;
- выбранный темп обучения;
- выбранная начальная точка (инициализация параметров);
- масштабирование признаков.

Из-за сложной зависимости и сложности самой функции потерь она может иметь несколько видов минимумов: **локальные** и **глобальные**.

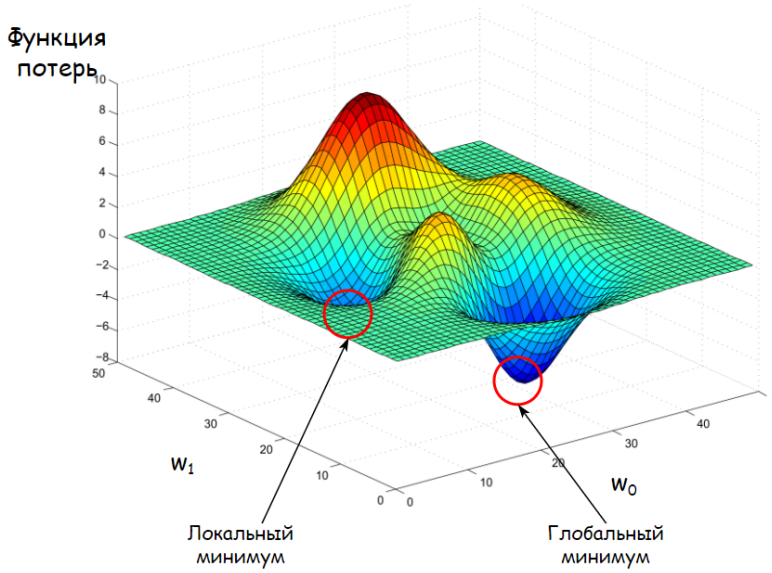
> **Локальный минимум** — это минимум на какой-то локальной области.

> **Глобальный минимум** — это минимум на всей области определения функции (на всём ландшафте).

Функция потерь с локальным и глобальным минимумом в случае одного параметра:



Функция потерь с локальным и глобальным минимумом в случае двух параметров:



Застряв в локальном минимуме, мы не найдём настоящие оптимальные значения параметров.

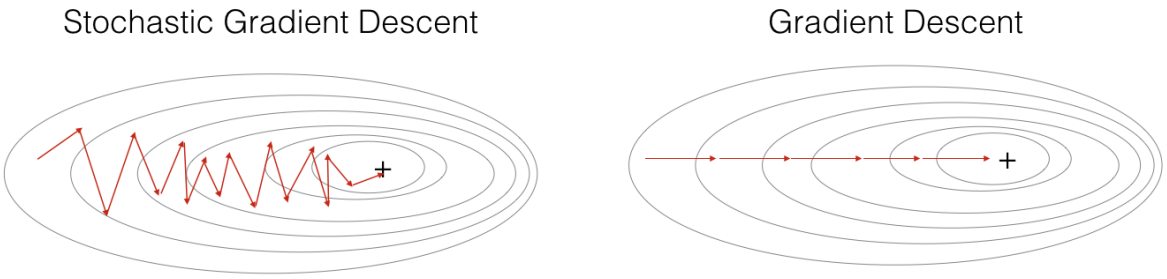
Чтобы частично решить эту проблему, используется не классический градиентный спуск, а его модификации, такие как **стохастический градиентный спуск**.

Стохастический градиентный спуск

> Стохастическая модификация предполагает, что один шаг градиентного спуска производится на основе градиента, рассчитанного не по всей выборке, а только по случайно выбранной части.

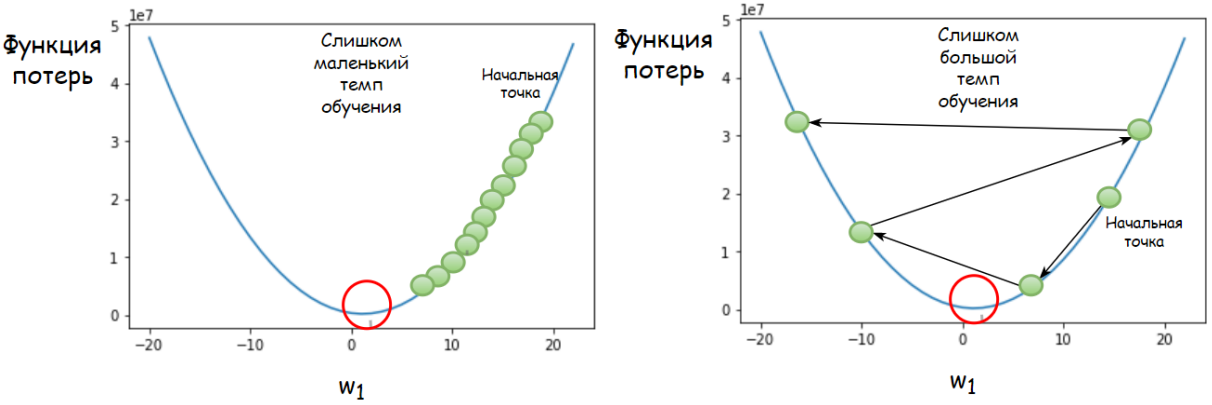
Благодаря этому вектор градиента всё время колеблется, и мы прыгаем из точки в точку, а не идём вдоль ровной линии, как это было в классическом градиентом спуске.

На рисунке ниже приведены графики «блуждания» точки в пространстве функции потерь (вид сверху).



Благодаря таким случайным колебаниям у нас повышается шанс «выкарабкаться» из локальных минимумов и дойти до глобального минимума.

Чтобы управлять шагами, как раз и существует параметр **темпа обучения**. Он позволяет управлять размером шага градиентного спуска.



Наиболее распространённые значения η <1: 0.01, 0.001 и т. д.

В реализации стохастического градиентного спуска в sklearn, с которым мы будем работать, именно такая идея и используется по умолчанию. Параметр η регулируется в процессе обучения — он уменьшается с ростом числа итераций по формуле:

$$\eta_t = \frac{\eta_0}{t^p},$$

где η_0 — начальное значение темпа обучения, p — мощность уменьшения (задаётся пользователем).

Еще один важный момент, на который стоит обратить внимание при работе с градиентным спуском — это обязательное **масштабирование факторов**

(приведение факторов к единому масштабу или к единым статистическим характеристикам), если их несколько.

Численное решение с помощью sklearn

Предварительная стандартизация:

```
from sklearn import preprocessing

#Инициализируем стандартизатор StandardScaler
scaler = preprocessing.StandardScaler()
#Производим стандартизацию
X_scaled = scaler.fit_transform(X)
#Составляем DataFrame из результата
X_scaled = pd.DataFrame(X_scaled, columns=features)
```

Обучение (поиск параметров) — метод `fit()`:

```
#Создаём объект класса линейной регрессии
sgd_lr = linear_model.SGDRegressor(random_state=42)
#Обучаем модель — ищем параметры по МНК
sgd_lr.fit(X_scaled , y)
```

Предсказание — метод `predict()`:

```
y_predict = sgd_lr.predict(X_scaled)
```

Посмотреть коэффициенты — атрибуты `coef_` и `intercept_`:

`sgd_lr .coef_` — коэффициенты при факторах
`sgd_lr.intercept _` — свободный член

Основные параметры SGDRegressor:

- `loss` — функция потерь. По умолчанию используется `"squared_loss"` — уже привычная нам MSE, но могут использоваться и несколько других.
- `max_iter` — максимальное количество итераций, выделенное на сходимость. По умолчанию 1000.

- `learning_rate` — режим управления темпом обучения. По умолчанию стоит `'invscaling'`. Этот режим уменьшает темп обучения по формуле, которую мы рассматривали ранее: $\eta_t = \frac{\eta_0}{t^p}$.
- `eta0` — начальное значение темпа обучения η_0. По умолчанию 0.01. Если параметр `learning_rate="constant"`, то значение этого параметра будет темпом обучения на протяжении всех итераций.
- `power_t` — значение мощности уменьшения p в формуле $\eta_t = \frac{\eta_0}{t^p}$. По умолчанию 0.25.

Сравнение аналитического и численного решений

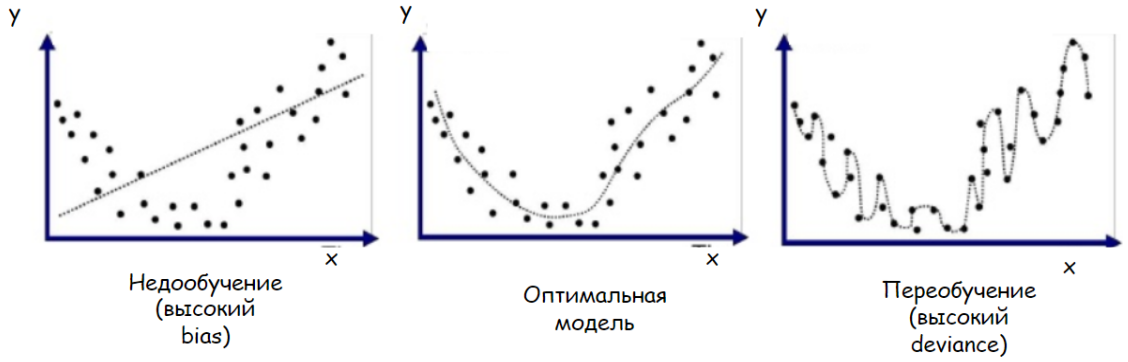
Показатель сравнения/Наименование в sklearn	LinearRegression	SGDRegressor
Метод решения и его сходимость к истинному минимуму	Аналитический — метод наименьших квадратов. Это главное преимущество метода: есть формула => подставили значения => совершили вычисления. Аналитический метод по определению является сходящимся, так как опирается на условие минимума функции.	Численный — метод стохастического градиентного спуска. Поиск минимума осуществляется итерациями. Сходимость зависит от множества факторов: темпа обучения, характера функции потерь, критерия остановки.
Функция потерь	Средний квадрат ошибки (MSE).	Любая гладкая функция, главное — чтобы она была дифференцируемой во всех точках. Функции потерь, доступные в sklearn, можно увидеть **здесь**. Каждая функция потерь предназначена для конкретной задачи.
Сложность алгоритма и время обучения	Кубическая сложность из-за вычисления обратной матрицы.	Линейная сложность, простые математические

	Время обучения кубически возрастает, что критически сказывается на наборах данных с большим количеством признаков.	операции умножения и сложения. Время обучения линейно возрастает с количеством признаков.
Возможность дообучения по новым данным	Отсутствует. Все данные должны быть поданы в модель заранее. Новый вызов `fit()` приведёт к новой настройке параметров.	Есть возможность дообучить модель на новых данных в режиме реального времени (**инкрементальное обучение**). Повторный вызов `fit()` уточняет уже существующие параметры модели.
Чувствительность к разному масштабу факторов	Низкая, стандартизация (нормализация) факторов желательна только на большом количестве признаков в данных.	Обязательная стандартизация (нормализация) факторов при наличии разных масштабов из-за особенностей сходимости.
Подбор внешних параметров	Внешних параметров нет.	Для поисков лучшего решения, возможно, придётся подбирать параметры: начальный темп обучения, режим обучения и т. д. Правильную реализацию подбора параметров мы обсудим в отдельном модуле.

Дилемма смещения и разброса

Переобучение (overfitting) — это проблема, когда модель может детально подстроиться под зависимость в обучающей выборке, но не уловить общей сути. Такая модель намного лучше работает с обучающими данными, чем с новыми.

Недообучение (underfitting) — это проблема, обратная переобучению. Модель из-за своей слабости не уловила никаких закономерностей в данных. В этом случае ошибка будет высокой как для тренировочных данных, так и для данных, не показанных во время обучения.



С теоретической точки зрения недообучение и переобучение характеризуются понятиями **смещения** и **разброса** модели.

Смещение (bias) — это математическое ожидание разности между истинным ответом и ответом, выданным моделью. То есть это ожидаемая ошибка модели.

$$bias(\hat{y}) = M[(y - \hat{y})]$$

Чем больше смещение, тем слабее модель. Если модель слабая, то она не в состоянии выучить закономерность. То есть налицо недообучение модели.

Разброс (variance) — это вариативность ошибки, то, насколько ошибка будет отличаться, если обучать модель на разных наборах данных. Математически это дисперсия (разброс) ответов модели.

$$variance(\hat{y}) = D[(y - \hat{y})]$$

Чем больше разброс, тем больше будет колебаться ошибка на разных наборах данных. Наличие высокого разброса и есть свидетельство переобучения: модель подстроилась под конкретный набор данных и даёт высокий разброс ответов на разных данных.

Разложение MSE на смещение и разброс:

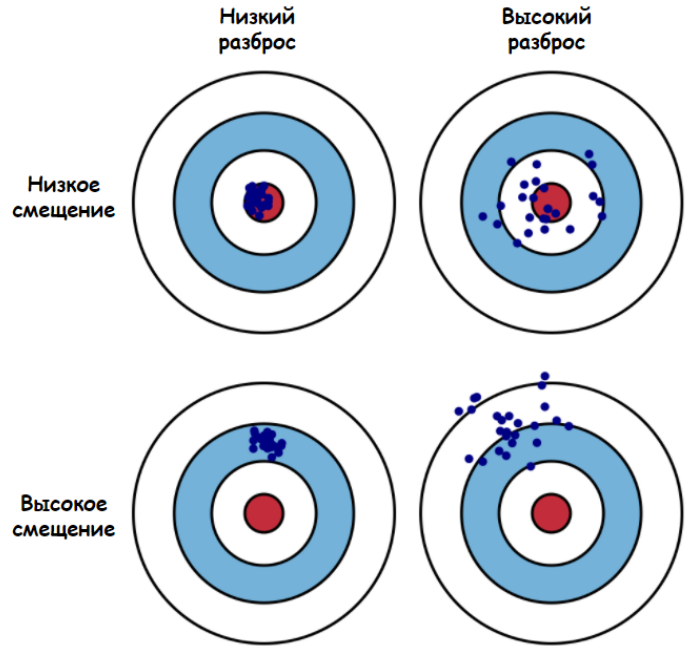
$$M[(y - \hat{y})^2] = bias(\hat{y})^2 + variance(\hat{y}) + \sigma^2,$$

где σ^2 — это неустранимая ошибка, вызванная случайностью, $bias(\hat{y})^2$ — смещение модели (в квадрате), $variance(\hat{y})$ — разброс модели.

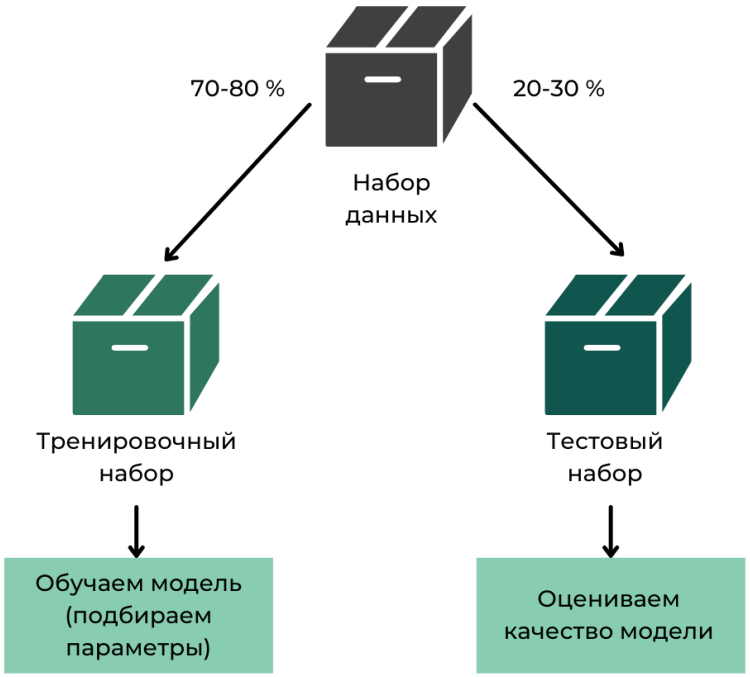
Дилемма смещения-дисперсии является центральной проблемой в обучении с учителем. В идеале мы хотим построить модель, которая точно описывает зависимости в тренировочных данных и хорошо работает на неизвестных данных. К сожалению, обычно это невозможно сделать одновременно.

Усложняя модель, мы пытаемся уменьшить смещение (bias), однако у нас появляется риск получить переобучение, то есть мы повышаем разброс (variance).

С другой стороны, снизить разброс (variance) позволяют более простые модели, не склонные к переобучению, но есть риск, что простая модель не уловить зависимостей и окажется недообученной, то есть мы повышаем смещение (bias).



Типичным решением является разделение данных на две части: **обучающий** и **тестовый наборы**.



Разделение выборки в sklearn

В sklearn для разделения выборки на тренировочную и тестовую есть функция [train_test_split()] из модуля model_selection. Данная функция принимает следующие **аргументы**:

- `X` и `y` — таблица с примерами и ответами к ним;
- `random_state` — зерно генератора случайных чисел;
- `test_size` — доля тестовой выборки. Определяет, в каких пропорциях будет разделена выборка. Стандартные значения — 70/30, 80/20.

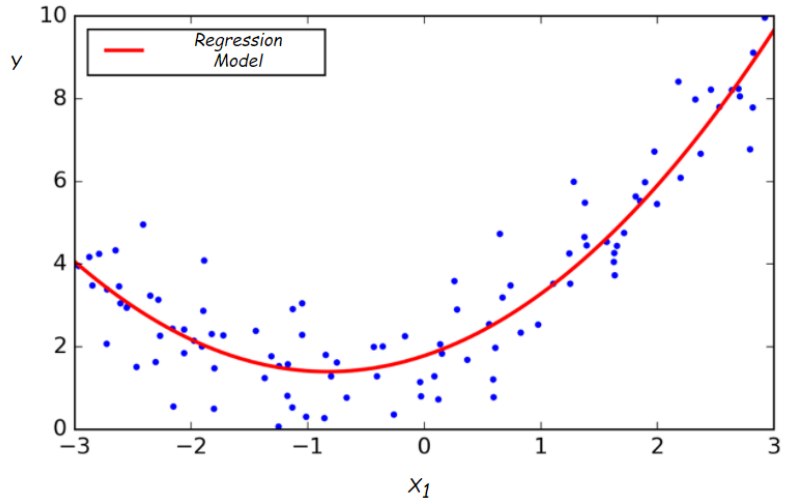
```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=40)
```

Полиномиальная регрессия

Полиномиальная регрессия (Polynomial Regression) — это более сложная модель, чем линейная регрессия. Вместо уравнения прямой в ней используется уравнение полинома (многочлена).

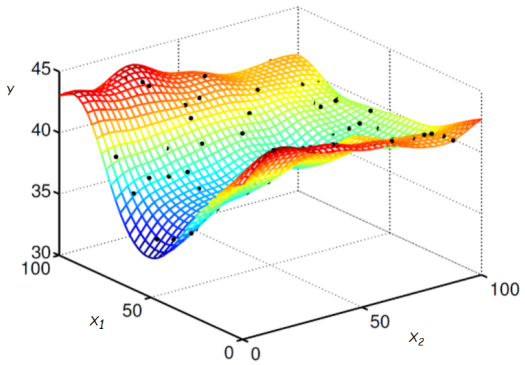
В простом двумерном случае (зависимость целевого признака от одного фактора) полином второй степени будет уравнением параболы:

$$\hat{y} = \omega_0 + \omega_1 x + \omega_2 x^2$$



Когда факторов больше одного, например два, то, помимо возведения фактора в квадрат, появляются ещё и комбинации из x_1 и x_2:

$$\hat{y} = \omega_0 + \omega_1 x_1 + \omega_2 x_2^2 + \omega_3 x_2 + \omega_4 x_2^2 + \omega_5 x_1 x_2$$



Полиномиальная регрессия в sklearn

```
#Создаём генератор полиномиальных признаков
poly = preprocessing.PolynomialFeatures(degree=2, include_bias=False)
poly.fit(X_train)
#Генерируем полиномиальные признаки для тренировочной выборки
X_train_poly = poly.transform(X_train)
#Генерируем полиномиальные признаки для тестовой выборки
X_test_poly = poly.transform(X_test)
#Создаём объект класса линейной регрессии
lr_model_poly = linear_model.LinearRegression()
#Обучаем модель по МНК
lr_model_poly.fit(X_train_poly, y_train)
#Делаем предсказание для тренировочной выборки
y_train_predict_poly = lr_model_poly.predict(X_train_poly)
#Делаем предсказание для тестовой выборки
y_test_predict_poly = lr_model_poly.predict(X_test_poly)
```

Регуляризация

> **Регуляризация** — это способ уменьшения переобучения моделей машинного обучения.

Мы намеренно пытаемся увеличить смещение модели, чтобы уменьшить разброс.

Методы регуляризации:

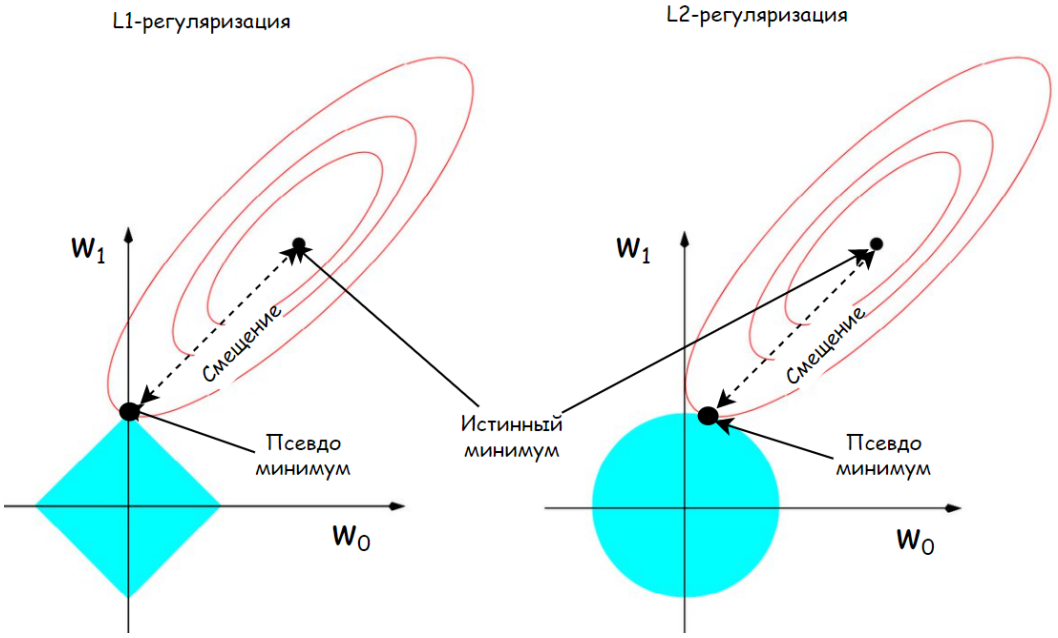
- **L1-регуляризация (Lasso)** — добавление к функции потерь суммы модулей коэффициентов, умноженных на коэффициент регуляризации α:

  $$L_1(\omega) = MSE + \alpha \sum_{j=1}^{m} |\omega_j|$$

- **L2-регуляризация (Ridge)**, или **регуляризация Тихонова** — добавление к функции потерь суммы квадратов коэффициентов, умноженных на коэффициент регуляризации α:

  $$L_2(\omega) = MSE + \alpha \sum_{j=1}^{m} (\omega_j)^2 \rightarrow min_{\omega}$$

Коэффициенты α (альфа) — это коэффициенты регуляризации. Они отвечает за то, насколько сильное смещение мы будем вносить в модель: чем оно больше, тем выше будет штраф за переобучение.



В результате добавления смещения мы находим не настоящий минимум, а псевдоминимум, который лежит на пересечении с ромбом при L1 (с окружностью — при L2). Такая оптимизация называется **условной**, или **оптимизацией с ограничениями**.

Регуляризация в sklearn

L1-регуляризация:

```
#Создаём объект класса линейной регрессии с L1-регуляризацией
lasso_lr_poly = linear_model.Lasso(alpha=0.1)
#Обучаем модель
lasso_lr_poly.fit(X_train_scaled_poly, y_train)
#Делаем предсказание для тренировочной выборки
y_train_predict_poly = lasso_lr_poly.predict(X_train_scaled_poly)
#Делаем предсказание для тестовой выборки
y_test_predict_poly = lasso_lr_poly.predict(X_test_scaled_poly)
```

L2-регуляризация:

```
#Создаём объект класса линейной регрессии с L2-регуляризацией
ridge_lr_poly = linear_model.Ridge(alpha=10)
#Обучаем модель
ridge_lr_poly.fit(X_train_scaled_poly, y_train)
#Делаем предсказание для тренировочной выборки
y_train_predict_poly = ridge_lr_poly.predict(X_train_scaled_poly)
#Делаем предсказание для тестовой выборки
y_test_predict_poly = ridge_lr_poly.predict(X_test_scaled_poly)
```

Помимо основных методов регуляризации L1 и L2, существует комплексный метод.

> **Эластичная сетка (ElasticNet)** — это комбинация из двух методов регуляризации.

Функция потерь в таком методе выглядит следующим образом:

$$L_2(\omega) = MSE + \alpha \cdot \lambda \sum_{i=1}^{m} |\omega_i| +$$

$$+ \alpha \cdot (1 - \lambda) \sum_{i=1}^{m} (\omega_i)^2 \rightarrow min_{\omega}$$

Параметры α и λ позволяют регулировать вклад L1- и L2-регуляризации. На практике данный метод используется гораздо реже, так как необходимо подбирать оптимальную комбинацию из двух параметров.