Линейная регрессия

1. Поиск градиента

Продолжить формулу для взятия векторной производной по x для функции квадрата ошибки (в красивом формате со всеми значками угловых скобок, операторов и т. д.). После выполнения этого вы вспомните формулу градиента для квадратичной ошибки и поймете как писать формулы на языке математической верстки TeX

$$f(x)=||Ax-y||^2=(Ax-y)^T(Ax-y)$$

$$D_f(x)=\dots$$

$$\nabla f(x)=\dots$$

2. Прямое решение через ноль производной

После пункта 1 у вас есть формула для градиента квадратичной функции ошибки. Она измеряет уровень "неверности" решения (вектора $\vec{x}$). Теперь её надо приравнять к 0 и построить формулу, чтобы узнать при каком x это происходит.

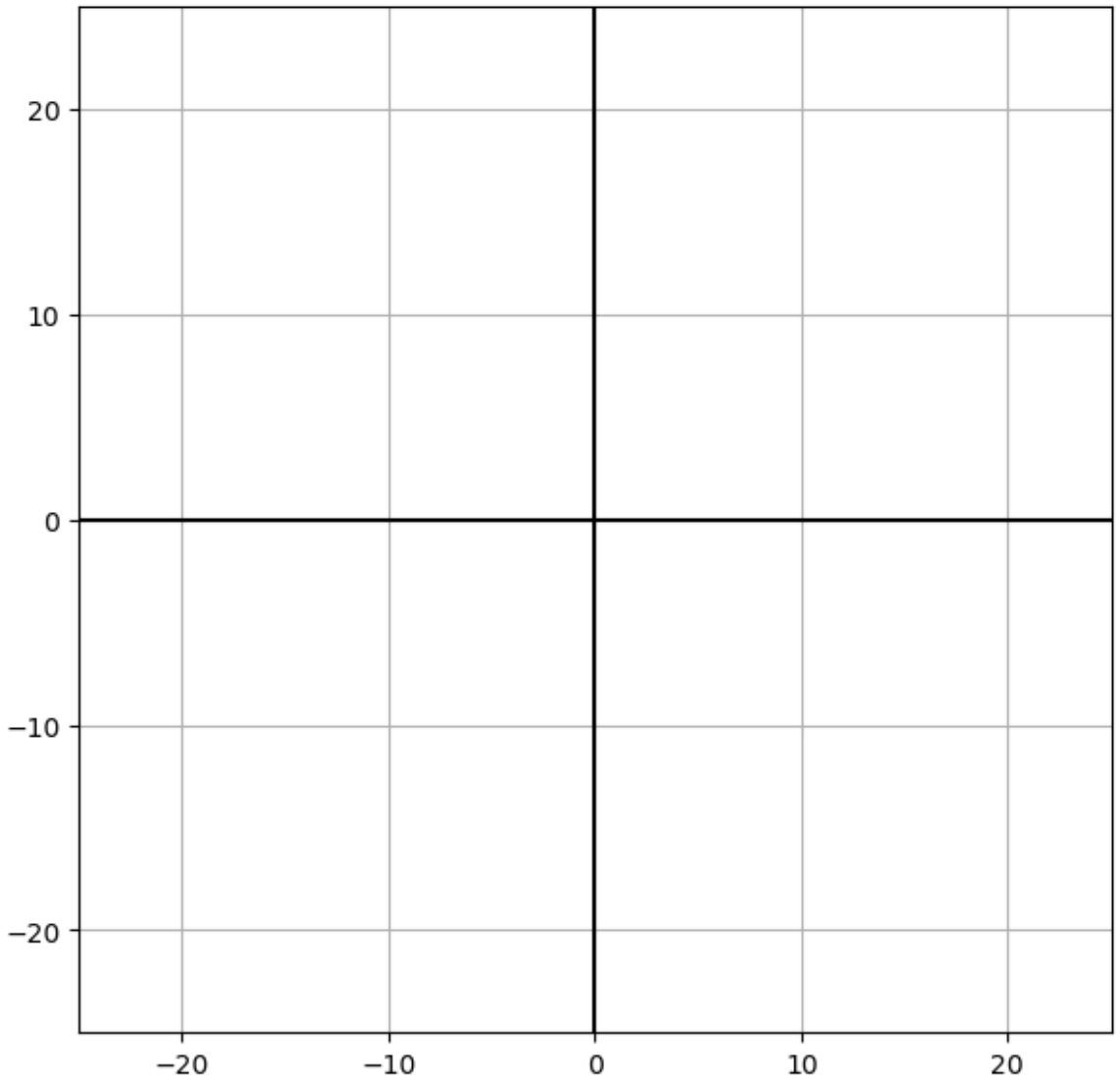
$$\nabla f(x)=\dots=0$$

$$x=\dots$$

2.1 Реализация прямого решения

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

# ваши параметры по вариантам
a_orig = ... # по формуле (-1)^n * 0.1 * n, n - номер в списке группы
b_orig = ... # по формуле n*(-1)^(n+1), где n - номер в списке группы
random_state = ... # x, где x - номер в списке группы

# это тот самый столбец который мы и должны отыскать будем нашими
методами.
# сейчас конечно мы его знаем наперед потому что нам надо создать
данные
# но в жизни мы изначально этих чисел не знаем и в жизни задача в том
чтобы их найти
x_orig = np.array([a_orig, b_orig])

np.random.seed(random_state)

A = np.stack([np.arange(0, 25), np.ones(25)]).T

# @ - операция матричного умножения в библиотеке NumPy
```

```
y = A @ x_orig + np.random.standard_normal(25)
# добавили нормально распределённого шума в данных
# этим система станет несовместна для обычного решения

f, ax = plt.subplots(figsize=(7, 7))

# A[:, 0] - выбрать первый столбец, A[0, :] - выбрать первую строку
plt.scatter(A[:, 0], y)

ax.set_xlim(-25, 25)
ax.set_ylim(-25, 25)
# рисуем координатные оси
ax.axvline(0, color="black")
ax.axhline(0, color="black")
ax.grid(True)
```



```
A, y
```

Вопрос - зачем был дописан вектор единиц справа к иксу?

ответ - ...

```
# допишите код для поиска a и b через 0 производной и выведите какой
вышел результат
a_b_analytical = ...
a_b_analytical

# постройте линию которая вышла рядом с изначальными данными


# изначальные данные
f, ax = plt.subplots(figsize=(7, 7))

plt.scatter(A[:, 0], y)
ax.set_xlim(-25, 25)
ax.set_ylim(-25, 25)
ax.axvline(0, color='black')
ax.axhline(0, color='black')
ax.grid(True)

# линия которая обучилась
# нужно вместо троеточий подсчитать значения y которые
# получаются при полученных параметрах линии
# в точках x1 = -25, x2 = 25, этим мы построим линию по 2 точкам
ax.plot([-25, 25], [..., ...])
```

2. Решение с помощью градиентного спуска

С помощью найденных выше формул градиента, совершить шаги градиентного спуска для тех же данных. Подобрать значение α, чтобы на графиках была видна сходимость за 10 шагов. Начинаем с параметров 0, 0

```
alpha = 1

a_b_analytical = np.array([0, 0])

for i in range(10):
    # получаем градиент
    grad = ...
    # получаем антиградиент
    antigrad = ...
    # обновляем наши параметры линии
    a_b_analytical += ...

    # рисуем результат текущего шага
```

```
    #  - изначальные данные
    f, ax = plt.subplots(figsize=(7, 7))

    plt.scatter(A[:, 0], y)
    ax.set_xlim(-25, 25)
    ax.set_ylim(-25, 25)
    ax.axvline(0, color='black')
    ax.axhline(0, color='black')
    ax.grid(True)

    # - линия которая обучилась
    ax.plot([-25, 25], [..., ...])
    ax.set_title(f'Линия по данным после шага {i+1}')
    plt.show()
```