Конспект по теме "Градиентный спуск"

Минимизация функции потерь

Чтобы сформулировать задачу для градиентного спуска, рассмотрим задачу обучения по-новому. Вы уже знакомы с функцией потерь. Она возвращает число потерь от неправильных ответов модели и обычно применяется для обучения.

В прошлом курсе мы приравнивали функцию потерь к *MSE*. В новой задаче обозначим функцию потерь как *L(y, a)* (от англ. *loss*, «потеря»), где вектор *y* — правильные ответы, вектор *a* — предсказания.

Задачу обучения модели через минимизацию функции потерь запишем так:

$$w = \arg\min_{w} L(a, y)$$

Чтобы вычислить *MSE*, или **квадратичную функцию потерь**, разность между правильными ответами и предсказаниями возводится в квадрат:

$$L(y, a) = \sum_{i=1}^{n} (a_i - y_i)^2$$

Какие ещё бывают функции потерь для задачи регрессии? Если в задаче важно, чтобы функция была менее чувствительна к выбросам, в отличие от *MSE*, применяют **абсолютную функцию потерь**, или знакомую вам *MAE*. Как и *MSE*, мы её рассматриваем как функцию потерь, а не как метрику качества.

$$L(y, a) = \sum_{i=1}^{n} |a_i - y_i|$$

В задаче классификации часто применяют метрику качества *accuracy*. Но как функция потерь она редко когда подходит. Всё потому, что у функции вычисления *accuracy* нет **производной**, которая показывает изменение функции при небольших изменениях аргумента.

Заменим *accuracy* **отрицательным логарифмическим правдоподобием**, или **логистической функцией потерь**. Функция потерь суммирует логарифмы вероятностей в зависимости от объекта. Если правильный ответ равен единице, прибавляется $log_{10}\ a_i$, а если нулю — то $log_{10}\ (1 - a_i)$. **Логарифм** — степень, в которую для извлечения аргумента нужно возвести основание (в нашем случае 10).

Запишем формулу:

$$L(y, a) = -\sum_{i=1}^{n} \begin{cases} \log_{10} a_i & \text{если } y_i = 1 \\ \log_{10}(1 - a_i) & \text{если } y_i = 0 \end{cases}$$

где a_i — вероятность класса 1 для объекта с индексом i. То есть значение a_i должно быть как можно выше у объекта положительного класса и ниже — у объекта отрицательного класса.

Название отрицательного логарифмического правдоподобия произошло от **функции правдоподобия**. Она вычисляет, с какой вероятностью модель ответит на все объекты правильно, если для ответа возьмёт значения a_i:

$$Likelihood(y, a) = \prod_{i=1}^{n} \begin{cases} a_i & \text{если } y_i = 1 \\ 1 - a_i & \text{если } y_i = 0 \end{cases}$$

Логарифмирование, то есть вычисление логарифма, применяется, чтобы «растянуть» значения на бóльший диапазон. Например, он был от 0 до 1, а стал от -∞ до 0. В таком диапазоне погрешности вычислений не важны. А умножение на -1 нужно, потому что итоговая функция потерь должна быть минимизирована.

В отличие от *accuracy*, логистическая функция потерь имеет производную.

Градиент функции

Минимум функции потерь не всегда можно найти вручную. Разобраться, в каком он направлении, поможет **градиент функции**.

Функция потерь зависит от параметров алгоритма. Эта функция **векторная** , то есть она принимает на вход вектор, а возвращает скаляр.

Градиент векторной функции — это вектор, состоящий из производных ответа по каждому аргументу. Обозначается символом ∇ (набла). Градиент функции f от n-мерного вектора x вычисляется так:

$$\nabla f(x) = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \dots, \frac{\partial f}{\partial x_n} \right)$$

где $\partial f / \partial x_i$ — производная функции f по аргументу x_i.Градиент функции от одного аргумента — это производная.

Градиент показывает направление самого быстрого роста функции. Но для задачи минимизации нам нужен противоположный вектор — направление наискорейшего убывания, то есть **антиградиент**:

$$-\nabla f(x)$$

Градиентный спуск

Градиентный спуск (*gradient descent*) — это итеративный алгоритм поиска минимума функции потерь. Он движется по направлению антиградиента и постепенно приближается к минимуму.

Добраться до минимума за одну итерацию сложно, ведь вектор антиградиента показывает направление убывания, а не конкретную точку минимума функции потерь.

Чтобы приступить к спуску, выберем отправное значение аргумента (вектор x). Оно называется **начальным значением** и обозначается x^0. С него будет сделан первый **шаг градиентного спуска**. Следующая точка x^1 образуется так: к точке x^0 прибавляется антиградиент, умноженный на размер шага градиентного спуска (μ).

$$x^1 = x^0 - \mu \times \nabla f(x)$$

Значение μ контролирует размер шагов градиентного спуска. Если шаг маленький, понадобится много итераций, но каждая приблизит нас к минимуму функции потерь. Когда шаг слишком большой, то минимум можно просто пропустить.

Таким же образом получим значения аргументов и на последующих итерациях. Номер итерации обозначим t. Чтобы получить новые значения x^t, к предыдущей точке прибавим антиградиент, умноженный на размер шага:

$$x^t = x^{t-1} - \mu \times \nabla f(x^{t-1})$$

Градиентный спуск завершается, когда:

- алгоритм прошёл заданное количество итераций

или

- значение аргумента x перестало меняться.

Градиентный спуск на Python

Резюмируем, что нужно сделать для запуска алгоритма градиентного спуска:

1. В аргументах алгоритма задать начальное значение x^0.
2. Рассчитать градиент функции потерь (это вектор производных по каждому аргументу, в который нужно передать вектор *x*).
3. Найти новое значение по формуле:

$$x^t = x^{t-1} - \mu \times \nabla f(x^{t-1})$$

   где *μ* — размер шага; задаётся в аргументах алгоритма.
4. Повторить заданное в аргументах число итераций.

```
import numpy as np

def func(x):
    # функция, которую нужно минимизировать

def gradient(x):
    # градиент функции func

def gradient_descent(initialization, step_size, iterations):
    x = initialization
    for i in range(iterations):
        x = x - step_size * gradient(x)
    return x
```