Линейная регрессия: ручная реализация против стандартной из sklearn

Часть 1. Теория

Линейная регрессия (англ. *Linear regression*) — это статистическая модель, описывающая зависимость одной переменной (зависимой) y от одной или нескольких других переменных (независимых) x.

Линейная регрессия имеет два случая: когда существует один признак x (одномерный случай) и когда признаков несколько, $d > 1$ (многомерный случай). Общая формула выглядит так:

$$\hat{y} = Xw$$

где

$\hat{y}$ - вектор предсказаний

X - матрица признаков размера $N \times (d + 1)$

w - вектор параметров длины $d + 1$

Однако, есть еще и запись с параметрами. Для одномерного случая

$$\hat{y} = w_0 + w_1 x$$

Для многомерного

$$\hat{y}_i = w_0 + w_1 x_{i1} + w_2 x_{i2} + \cdots + w_d x_{id}$$

Здесь, мы будем рассматривать одномерный случай, то есть вектор предсказаний вида $\hat{y} = w_0 + w_1 x$. В учебных целях удобно рассматривать одномерный случай, поскольку он проще и нагляднее. Однако в реальных задачах число признаков может быть очень большим.

Что мы пытаемся сделать? Мы пытаемся найти такую прямую (или гиперплоскость), которая лучше всего описывает зависимость между признаками и целевой переменной. То есть, мы должны найти такие правильные веса параметров $w_0, w_1, ..., w_d$ при которых ошибка предсказания минимальна. Как это сделать?

Функция ошибки (Loss function)

Чтобы понять, какие веса $w_0, w_1, ..., w_d$ являются "лучшими", нам нужно измерить, насколько хорошо модель делает предсказания. Для этого используют функцию ошибки (loss function).

Наиболее распространённая мера ошибки в линейной регрессии - среднеквадратичная ошибка (MSE):

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

y_i - реальные значения

$\hat{y}_i$ - значения предсказанные моделью

Чем меньше $J(w)$, тем лучше. В нашем случае мы должны минимизировать функцию ошибку. Для одномерного случая, функция ошибки выглядит так:

$$J(w) = \frac{1}{N} \sum_{i=1}^{N} \big(y_i - (w_0 + w_1 x_i)\big)^2$$

Минимизация функции ошибки

Способ 1.

Можно взять частную производную функции ошибки по каждому весу w_i, приравнять их нулю и решить систему уравнений. В результате получается формула:

$$w = (X^\top X)^{-1} X^\top y$$

> 💡 Вывод нормального уравнения (least squares normal equation)
>
> $$J(w, X, y) = \frac{1}{2n} \sum_{i=1}^{n} (w^T x_i - y_i)^2 = \frac{1}{2n} \|Xw - y\|^2 = \frac{1}{2n} (Xw - y)^T (Xw - y) = \frac{1}{2n} \big((Xw)^T Xw - (Xw)^T y - y^T Xw + y^T y\big) = \frac{1}{2n} \big(w^T X^T Xw - 2y^T Xw + y^T y\big) \frac{\partial J}{\partial w}$$

Это общая матричная формула для любого числа признаков. Для нашего случая, когда у нас только один признак x, будет:

$$w_1 = \frac{\sum_{i=1}^{N} (x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^{N} (x_i - \bar{x})^2}$$

$$w_0 = \bar{y} - w_1 \bar{x}$$

где:

$\bar{x}$ - среднее значение признака

$\bar{y}$ - среднее значение целевой переменной

Главный недостаток данного способа заключается в том, что вычисление становится сложным, если признаков слишком много!

Способ 2. Градиентный спуск (Gradient Descent).

Аналитическое решение даёт точный ответ, но оно неэффективно при большом числе признаков. Альтернативный подход - итеративный метод градиентного спуска.

Идея: Мы начинаем с произвольных весов w (обычно начинают с нуля) и постепенно «шагаем» в сторону уменьшения ошибки.

Функция ошибки для одномерного случая:

$$J(w) = \frac{1}{N} \sum_{i=1}^{N} \big(y_i - (w_0 + w_1 x_i)\big)^2$$

Градиенты:

$$\frac{\partial J}{\partial w_0} = -\frac{2}{N} \sum_{i=1}^{N} \big(y_i - (w_0 + w_1 x_i)\big)$$

$$\frac{\partial J}{\partial w_1} = -\frac{2}{N}\sum_{i=1}^{N} x_i(y_i - (w_0 + w_1 x_i))$$

Обновление весов

$$w_j := w_j - \eta \frac{\partial J}{\partial w_j}$$

где:

η - скорость обучения (learning rate)

То есть каждый шаг уменьшает ошибку, двигая веса в сторону противоположную градиенту. Обычно, η = 0.01.

Когда останавливать градиентный спуск? Или же, сколько нужно итераций?

1. Веса почти перестали обновляться

$$|w_j^{(t)} - w_j^{(t-1)}| < 10^{-6}$$

2. Ошибка почти не меняется

$$|J^{(t)} - J^{(t-1)}| < \varepsilon$$

где ε - маленькое число

Алгоритм градиентного спуска (одномерный случай)

```
1. Инициализируем w0 = 0, w1 = 0
2. Выбираем скорость обучения η # например, 0.01
3. Для t = 1 ... T: # цикл
    y_pred = w0 + w1 * x # модель
    grad_w0 = -(2/N) * Σ(y - y_pred) # частная производная w0
    grad_w1 = -(2/N) * Σ(x * (y - y_pred)) # частная производная w1
    w0 = w0 - η * grad_w0 # обновление веса w0
    w1 = w1 - η * grad_w1 # обновление веса w1
4. Возвращаем итоговые w0 и w1 # итог
```

Градиентный спуск работает, потому что функция ошибки линейной регрессии - выпуклая. У нее один глобальный минимум, если выбрать правильный шаг - то рано или поздно он найдет верные веса.

Часть 2. Реализация

Сейчас, я бы хотел показать простую реализацию линейной регрессии чистым Python и с *sklern*. Прежде всего, в качестве простого датасета, возьмем:

```
data = [
    (1, 8.3), (2, 11.1), (3, 14.2), (4, 17.3), (5, 19.8),
    (6, 23.1), (7, 25.2), (8, 29.0), (9, 31.4), (10, 34.1),
    (11, 38.2), (12, 40.5), (13, 44.1), (14, 47.0), (15, 50.3),
    (16, 53.1), (17, 56.2), (18, 59.0), (19, 62.4), (20, 65.1),
    (21, 68.3), (22, 71.2), (23, 74.0), (24, 77.4), (25, 80.1),
    (26, 83.5), (27, 86.0), (28, 89.2), (29, 92.3), (30, 95.0),
    (31, 98.4), (32, 101.3), (33, 104.1), (34, 107.5), (35, 110.2),
    (36, 113.1), (37, 116.4), (38, 119.2), (39, 122.3), (40, 125.0),
    (41, 128.4), (42, 131.3), (43, 134.1), (44, 137.4), (45, 140.2),
    (46, 143.5), (47, 146.0), (48, 149.1), (49, 152.2), (50, 155.1)
]
```

Аналитическое решение

Найдем средние x, y

```
N = len(data)

def mean_x(data):
    return sum(x for x, y in data) / N

def mean_y(data):
    return sum(y for x, y in data) / N

x_mean = mean_x(data)
y_mean = mean_y(data)
```

Далее, будем находить

$$w_1 = \frac{\sum_{i=1}^{N}(x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^{N}(x_i - \bar{x})^2}$$

Заметим, что в числителе у нас $Cov(x, y)$ а в знаменателе $Var(x)$, то есть

$$w_1 = \frac{Cov(x, y)}{Var(x)}$$

```
def covariance_xy(data):
    return sum((x - x_mean) * (y - y_mean) for x, y in data)


def variance_x(data):
    return sum((x - x_mean) ** 2 for x, y in data)


w1 = covariance_xy(data) / variance_x(data)
```

Далее, найдем параметр w_0

$$w_0 = \bar{y} - w_1\bar{x}$$

```
w0 = y_mean - w1 * x_mean

print(f"Модель: y = {w0} + {w1} * x")
```

Полный код:

```
N = len(data)

def mean_x(data):
    return sum(x for x, y in data) / N

def mean_y(data):
    return sum(y for x, y in data) / N

x_mean = mean_x(data)
y_mean = mean_y(data)


def covariance_xy(data):
    return sum((x - x_mean) * (y - y_mean) for x, y in data)


def variance_x(data):
    return sum((x - x_mean) ** 2 for x, y in data)


w1 = covariance_xy(data) / variance_x(data)
w0 = y_mean - w1 * x_mean

print(f"Модель: y = {w0} + {w1} * x")
```

Вывод:

```
Модель: y = 4.928081632653061 + 3.0076830732292916 * x
```

При x = 51, мы получаем y = 158.319918367

Решение с помощью градиентного спуска

Задаём скорость обучения, число итераций и инициализируем параметры

```
eta = 0.0001
iterations = 400000

w0 = 0.0
w1 = 0.0

n = len(data)
```

Градиентный спуск

```
for i in range(iterations):
    dw0 = 0
    dw1 = 0

    for x, y in data:
        error = (w0 + w1 * x) - y
        dw0 += error
        dw1 += error * x

    dw0 *= 2 / n
    dw1 *= 2 / n

    w0 -= eta * dw0
    w1 -= eta * d
```

Полный код:

```
eta = 0.0001
iterations = 400000

w0 = 0.0
w1 = 0.0

n = len(data)



for i in range(iterations):
    dw0 = 0
    dw1 = 0

    for x, y in data:
        error = (w0 + w1 * x) - y
        dw0 += error
```

```
        dw1 += error * x

    dw0 *= 2 / n
    dw1 *= 2 / n

    w0 -= eta * dw0
    w1 -= eta * dw1

print("w0 =", w0)
print("w1 =", w1)
print("Модель: y =", w0, "+", w1, "* x")
```

Вывод:

```
w0 = 4.928081614301114
w1 = 3.007683073774553
Модель: y = 4.928081614301114 + 3.007683073774553 * x
```

При x = 51, мы получаем y = 158.3199183768033

Значение, очень близко похоже к аналитическому решению, однако если мы бы поставили шаг = 4000, результат был хуже. Поэтому важно выбирать правильный шаг.

Решение линейной регрессии с помощью библиотеки scikit-learn

Импорт необходимых библиотек

```
from sklearn.linear_model import LinearRegression
import numpy as np
```

Преобразуем данные в нужный формат

```
X = np.array([x for x, y in data]).reshape(-1, 1)
y = np.array([y for x, y in data])
```

Обучаем модель и извлекаем параметры

```
model = LinearRegression().fit(X, y)

w0 = model.intercept_
w1 = model.coef_[0]

print(f'{w0} + {w1}x')
```

Полный код:

```
from sklearn.linear_model import LinearRegression
import numpy as np


X = np.array([x for x, y in data]).reshape(-1, 1)
y = np.array([y for x, y in data])

model = LinearRegression().fit(X, y)

w0 = model.intercept_
w1 = model.coef_[0]

print(f'Модель: y = {w0} + {w1} * x')
```

Вывод:

```
Модель: y = 4.928081632653047 + 3.007683073229291 6 * x
```

При x = 51, мы получаем y = 158.319918367