Кантонистова Елена Олеговна
Математика для анализа данных

Содержание

1. Лекция 1

1.1. Как устроен курс

- 1 модуль - матан и линал
- 2 модуль - дискра, тервер
- 3/4 модуль - статистика

1.2. Практика

Исследовать функцию

$$f(x) = x^3 - 3x^2 + 4$$

Найдём

1.3. SymPy

В Питоне есть библиотека SymPy, которая предоставляет интерфейс для вычисления производных

```
!pip install sympy
```

Далее в питоне зададим переменную и производную:

```
import sympy as sp
x = sp.Symbol('x')
sp.diff(x**6)
```

Теперь будем анализировать функцию из практики:

```
def f(x):
    return x**3 - 3*x**2 + 4
```

Чтобы найти нули функции, надо решить уравнение $f(x) = 0$. В SymPy для этого есть функция solve:

```
sp.solve(f(x), x)
[-1, 2]
```

Теперь найдём производную функции $f(x)$ и затем её нули, чтобы найти экстремумы

```
df_x = sp.diff(f(x))
#df_x == 3x^2

sp.solve(df_x, x)
# [0, 2]

f(0), f(2)
# (4, 0)
```

Точно также очень просто можем находить втору производную и находить точки перегиба функции

```
d2f_x = sp.diff(df_x)
d2f_x
# 6x - 6

sp.solve(d2f_x, x)
#[1]

f(1)
#2
```

1.4. Рисуем графики

Что нам нужно будет сделать?

- Нарисовать график $f(x)$, подписать оси
- НАпечатать под графиком при помощи *Markdown* экстремумы, точки перегиба и значения функции f(x) в этих точках

1.4.1. Подключаем библиотеку

```
import matplotlib.pyplot as plt
%matplotlib inline #      -
```

1.4.2. Рисуем график функции f(x)

```
import numpy as np

x_values = [x for x in np.arange(-5, 5, 0.1)]
#  or     = np.linspace(-5, 5, 100)
f_values = [f(x)] for x in x_values]

plt.plot(x_values, f_values)
```

Вставить график

1.4.3. Изменяем размер картинки и граничные значения по осям

```
plt.figure(figsize=(10, 10))

plt.plot(x_values, y_values)

plt.xlim([-3, 5])
plt.ylim([-5, 7])
```

Вставить график

1.4.4. Добавим названия осей и подпись к графику

```
plt.title('Graph of fucntion f(x) with extremum and dots of ...')

plt.xlabel('x')
plt.ylabel('f(x)')
```

Вставить график

1.4.5. Добавим оси и подпишем под картинкой информацию об экстремумах и точках перегиба

```
import numpy as np

x_values = [x for x in np.arange(-5, 5, 0.1)]
f_values = [f(x) for x in x_values]

plt.figure(figsize=(10,10))

plt.axvline(x=0, c = 'black')
plt.axhline(y=0, c = 'black')

plt.plot(x_values, f_values)

plt.xlim([-3, 5])
plt.ylim([-5, 7])

plt.title('Graph of fucntion f(x) with extremum and dots of ...')

plt.xlabel('x')
plt.ylabel('f(x)')

plt.show()
```

Вставить график

1.5. Метод градиентного спуска

1.5.1. Теорема о градиенте

Градиент - это вектор, в направлении которого функция растёт быстрее всего.
Антиградиент (вектор противоположный градиенту) - вектор, в направлении которого функция быстрее всего убывает.

1.5.2. Применение в машинном обучении

Для чего нам это нужно? В машинном обучении мы минимизируем значение функции, которая показывает ошибку модели. Иными словами: наша задача при обучении модели - найти такие веса **w**, на которых достигается **минимум функции ошибок**.
В простейшем случае, если ошибка среднеквадратическая, то её график - парабола.

1.5.3. Идея применения градиентного спуска

На каждом шаге (на каждой итерации метода) движемся в сторону антиградиента функции потерь!
То есть на каждом шаге движемся в направлении уменьшении ошибки.
Вектор градиента функции потерь обозначают **grad Q** или ∇Q

1.5.4. Метод градиентного спуска на пальцах

- Встаём в некоторую точку функции
- Вычисляем градиент
- Переходим в новую точку в направлении антиградиента
- Повторяем процесс из новой точки

1.5.5. Метод градиентного спуска (одномерный случай)

Пусть у нас только один вес - w.
Тогда при добавлении к весу w слагаемоего $-\frac{\partial Q}{\partial w}$ функция $Q(w)$ убывает.
Тогда алгоритм выглядит следующим образом:

- Инициализируем вес $w^{(0)}$
- На каждом следующем шаге обновляем вес, добавляя $-\frac{\partial Q}{\partial w}(w^{(k-1)})$:

$$w^{(k)} = w^{(k-1)} - \frac{\partial Q}{\partial w}(w^{(k-1)})$$

1.5.6. Метод градиентного спуска (общий случай)

Пусть $w_0,\ w_1,\ \dots\ ,\ w_n$ - веса, которые мы ищем.
Тогда $\nabla Q(w) = \{\frac{\partial Q}{\partial w_0},\ \frac{\partial Q}{\partial w_1},\ \dots\ ,\ \frac{\partial Q}{\partial w_n}\}$
Тогда алгоритм выглядит так:

- Инициализируем веса $w^{(0)}$ (заметим, что это вектор весов)
- На каждом шаге обновляем веса по формуле:

$$w^{(k)} = w^{(k-1)} - \nabla Q(w^{(k-1)})$$

1.5.7. Параметр learning rate

В формулу обычно добавляют параметр η - величина градиентного спуска (**learning rate**). Он отвечает за скорость движения в сторону антиградиента:

- Инициализируем веса $w^{(0)}$ (заметим, что это вектор весов)
- На каждом шаге обновляем веса по формуле:

$$w^{(k)} = w^{(k-1)} - \eta\nabla Q(w^{(k-1)})$$

1.5.8. Теорема о поиске в выпуклой гладкой функции

Если функция $Q(w)$ выпуклая и гладкая, а также имеет минимум в точке w^*, то метод градиентного спуска при аккуратно подобранному η через некоторое число шагов гарантированно попадает в малую окрестность точки w^*.

1.6. Реализация градиентного спуска на python

```
def gradient_descent(x_start, learning_rate, epsilon, num_iterations):
    x_curr = x_start
    df_x = sp.diff(f(x))

    trace = []
    trace.append(x_curr)

    for i in range(num_iterations):
        x_new = c_curr + df_x.subs(x, x_curr)
        trace.append(x_new)

        if abs(x_new - x_curr) < epsilon:
            return x_curr, trace

    return x_curr, trace
```

2. Лекция 2

2.1. Линейная регрессия

Линейная регрессия - функция $a(x) = \omega_0 + \omega_1 x_1 + \omega_2 x_2 + \cdots + \omega_l x_l$, где x - **вектор признаков**
Также есть **целевая переменная**, которую мы предсказываем - y
w - **веса** линейной регрессии

Запишем в другой форме: $a(x) = \omega_0 + \sum\limits_{i=1}^{l} \omega_i x_i$

Также мы это можем записать в другой форме: давайте добавим ещё один признак у всех обектов, который будет равен единице: $x = (1, x_1, x_2, \ldots, x_n)$, и тогда всё записывается ещё красивее:

$$a(x) = \sum_{i=0}^{l} \omega_i x_i = (\overrightarrow{\omega}, \overrightarrow{x})$$

Где $a(x) = (\overrightarrow{\omega}, \overrightarrow{x})$ - предсказание модели на объекте x. Но это предсказание для одного объекта.
Мы можем записать предсказания в матричном виде для нескольких объектов. Возьмём X - матрицу объект-признак. В каждом строке описан один объект, а кол-во строк - это кол-во объектов.
В матричном виде предсказание выглядит как $a(X) = X \cdot w$

2.2. Функция ошибок MSE

MSE (Mean Squared Error) $= \frac{1}{d} \sum\limits_{i=1}^{d} (a(x_i) - y_i)^2$, где d - количество данных. При этом в задаче обучения мы хотим $MSE \to \min\limits_{\overrightarrow{\omega}}$. Мы уже научились и можем делать минимизацию функции с помощью Градиентного спуска.

2.3. Матричное представление MSE

$$MSE = \frac{1}{d} ||X\omega - y||^2$$

Где $||\overrightarrow{a}|| = \sqrt{a_1^2 + a_2^2 + \cdots + a_n^2}$
Соответственно $||\overrightarrow{a}||^2 = a_1^2 + a_2^2 + \cdots + a_n^2$

2.4. Как решать задачу минимизации MSE

2.4.1. Аналитическое решение

Это решение, которое даёт точное решение
Решаем уравнение $\nabla_\omega Q(\omega) = 0$

2.4.2. Приближённое решение

С помощью GD шагаем $\omega = \omega - \eta \nabla_\omega Q(\omega)$

2.5. Производная по вектору

Пусть у нас есть $\overrightarrow{x} = (x_1, \ldots, x_n)$
Градиент функции $f(x)$ рассчитывается как $\nabla_x f(x) = (\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{\partial x_n})$

Если мы хотим взять градиент по матрице A от функции f, то это выглядит как $\nabla_A f(A) = \begin{pmatrix} \frac{\partial f}{\partial A_{11}} & \cdots & \frac{\partial f}{\partial A_{1n}} \\ \cdots & \cdots & \cdots \\ \frac{\partial f}{\partial A_{n1}} & \cdots & \frac{\partial f}{\partial A_{nn}} \end{pmatrix}$

2.5.1. Пример 1. Подсчёт градиента скалярного произведения

Пусть у нас есть вектор весов $\overrightarrow{\omega}$ и вектор $\overrightarrow{x}$. Есть скалярное произведение $(\overrightarrow{\omega}, \overrightarrow{x})$. Мы хотим посчитать $\nabla_x(\overrightarrow{\omega}, \overrightarrow{x})$.
Мы знаем, что $\frac{\partial}{\partial x_i}(\overrightarrow{\omega}, \overrightarrow{x}) = \frac{\partial}{\partial x_i}(\omega_1 x_1 + \cdots + \omega_n x_n) = \omega_i$
Тогда $\nabla_x(\overrightarrow{\omega}, \overrightarrow{x}) = (\frac{\partial}{\partial x_1}(\overrightarrow{\omega}, \overrightarrow{x}), \frac{\partial}{\partial x_2}(\overrightarrow{\omega}, \overrightarrow{x}), \ldots, \frac{\partial}{\partial x_n}(\overrightarrow{\omega}, \overrightarrow{x})) = (\omega_1, \omega_2, \ldots, \omega_n) = \overrightarrow{\omega}$

2.5.2. Пример 2. Подсчёт градиента от матрицы

Пусть есть матрица $A_{n\times n}$ и вектор $\overrightarrow{x} \in \mathbb{R}^n$.
Функция x^TAx - это число (давайте посмотрим на размерности)
$\underline{(1\times n)(n\times n)}(n\times 1) = (1\times n)(n\times 1) = (1\times 1)$
Теперь мы хотим от этой функции находить градиент:
Давайте попробуем посчитать $\frac{\partial}{\partial x_i}x^TAx = \frac{\partial}{\partial x}\sum\limits_{j=1}^{n} x_j(Ax)_j = \frac{\partial}{\partial x_i}\sum\limits_{j=1}^{n} x_j(\sum\limits_{k=1}^{n} a_{jk}x_k) = \frac{\partial}{\partial x_i}\sum\limits_{j=1}^{n}\sum\limits_{k=1}^{n} a_{jk}x_jx_k =$
$\sum\limits_{j=1,j\neq i}^{n} a_{ji}x_j + \sum\limits_{k=1,k\neq i}^{n} a_{ik}x_i + 2a_{ii}x_i = \sum\limits_{i=1}^{n}\sum\limits_{j=1}^{n}(a_{ij}+a_{ji})x_j$ - i-я производная.

Тогда $\nabla_x(x^TAx) = (A + A^T)x$

2.6. Минимизация MSE с помощью градиента

Вспоминаем, что MSE выглядит как $||y - X\omega||^2$. Но это можно переписать в явном виде без квадрата:
$||y - X\omega||^2 = (y - X\omega)^T(y - X\omega) \to \min\limits_{\omega}$
Раскрываем скобки для поиска градиента: $\nabla_\omega((y^Ty)^{=0} - \omega^TX^Ty - y^TX\omega + \omega^TX^TX\omega) = 0$
$\nabla_\omega(-\omega^TX^Ty - y^TX\omega + \omega^TX^TX\omega) = -X^Ty - X^Ty + 2X^TX\omega = 0$ (для последнего слагаемого смотрим вывод пункта 2.5.2)
Перекинем слагаемые в разные стороны: $2X^TX\omega = 2X^Ty$. Сократим на двойку. Мы бы могли сократить матрицы, но обратной может не быть. Зато мы можем с каждой из сторон умножить на обратную матрицу к X^TX:
$X^TX\omega = X^Ty \to (X^TX)^{-1}(X^TX)\omega = (X^TX)^{-1}X^Ty \to \omega = (X^TX)^{-1}X^Ty$

2.7. Градиент функции потерь

Из подсчитанного можем сказать, что градиент функции потерь для MSE будет выглядеть как

$$\nabla Q(\omega) = 2X^T(X\omega - y)$$

Но давайте будем находить ω с помощью GD:

- На шаге обновления точки, у нас $\omega_{next} = \omega_{prev} - \eta\nabla Q(\omega)$
- Запишем зная, чему равно $\nabla Q(\omega)$: $\omega_{next} = \omega_{prev} - 2\eta X^T(X\omega_{prev} - y)$

2.8. Реализация на питоне

2.8.1. Генерирование данных для задачи регрессии

Давайте сгенирируем данные и визиализируем их:

```
import random
import matplotlib.pyplot as plt
import numpy as np
%matplotlib inline

X = np.linspace(-10, 10, 100)

print(X.shape)

y = X * (np.random.random_sample(len(X)) + 0.5)
X = X.reshape(len(X), 1)

print(X.shape)

plt.scatter(X, y)
```

2.8.2. Функция подсчёта ошибки

Также давайте напишем свой MSE:

```
def MSE(X, y, theta):
    m = len(y)

    error = (1./m) * (np.linalg.norm(X @ theta - y) ** 2)
    return error
```

2.8.3. Реализация градиентного спуска

Теперь у нас есть всё, чтобы реализовать свой градиентный спуск:

```
def gradient_descent(X, y, learning_rate, iterations):

    X = np.hstack((np.ones((X.shape[0], 1)), X)) # add column of ones
    params = np.random.rand(X.shape[1])

    m = X.shape[0]

    cost_track = np.zeros((iterations, 1))

    for i in range(iterations):
        params = params - 2./m * learning_rate * (X.T @ ((X @ params) - y))
        cost_track[i] = MSE(X, y, params)

    return cost_track, params
```

2.8.4. Функция предсказания модели

Записать предсказание модели можно очень просто:

```
def predict(X, params):
    X = np.hstack((np.ones((X.shape[0], 1)), X))
    return X @ params
```

2.8.5. Применение градиентного спуска

Применяем градиентный спуск:

```
track, weights = stochastic_gradient_descent(X, y, 0.01, 100)
plt.plot(track) # visualize errors
```

Теперь сделаем предсказание и посмотрим на визуализацию этого предсказания:

```
pred = predict(X, weights)
plt.scatter(X, y)
plt.plot(X, pred, '-', c = 'r')
```

3. Лекция 3

3.0.1. Один из недостатков градиентного спуска

С точки зрения реализации есть следующий недостаток:

На каждом шаге вычисления $\nabla Q(w)$ мы вычисляем производную по каждому весу от каждого объекта. То есть мы вычисляем целую матрицу производных - это затратно и по времени, и по памяти

3.1. Стохастический градиентный спуск

Stochastic Gradient Descent

На каждом шаге мы выбираем **один случайный объект** и сдвигаемся в сторону антиградиента по этому объекту:

$$\omega^{(k)} = \omega^{(k-1)} - \eta_k \cdot \nabla q_{i_k}(\omega^{(k-1)})$$

где $\nabla q_{i_k}(\omega)$ - градиент функции, вычисленный только по объекту с номером i_k (а не по всей обучающей выборке)

Если функция $q(\omega)$ выпуклая и гладкая, а также имеет минимум в точке ω^*, то метод стохастического градиентного спуска при аккуратно подобраном η (LR) через некоторое число шагов гарантированно попадает в малую окрестность точки ω^*. Однако сходится метод медленнее, чем обычный градиентный спуск.

3.2. Mini-Batch Gradient Descent

Промежуточное решение между классическим градиентным спуском и стохастическим вариантом

- Выбираем batch size (например, 32, 64, и т.д.). Разбиваем все пары объект-ответ на группы размера batch size
- На i-й итерации градиентного спуска вычисляем $\nabla Q(\omega)$ только по объектам i-го батча:

  $$\omega^{(k)} = \omega^{(k-1)} - \eta_k \nabla Q_i(\omega^{(k-1)})$$

  где $\nabla Q_i(\omega^{(k-1)})$ - градиент функции потерь, вычисленный по объектам из i-го батча

3.3. Ещё один недостаток градиентного спуска

Мы можем застрять в локальном минимуме и не дойти до глобального минимума

3.4. Метод моментов (Momentum)

Будем добавлять какие-то предыдущие значения шагов, которые будут аналогом инерции из физики:

Вектор инерции (усреднение градиента по предыдущим шагам):

$$h_0 = 0$$

$$h_k = \alpha h_{k-1} + \eta \nabla Q(w^{k-1})$$

Формула метода моментов:

$$w^{(k)} = w^{(k-1)} - h_k$$

Подробнее:

$$w^{(k)} = w^{(k-1)} - \eta \nabla Q(w^{(k-1)}) - \alpha h_{k-1}$$

3.5. Преимущества метода моментов

Проще подбирать параметр α, чем делать несколько запусков обычного градиентного спуска, потому что в многомерном случае сложнее генерировать эти данные и сложнее так находить глобальный минимум

Также иногда, когда у нас данных много, GD может работать неделю, и нам непозволительно делать много запусков

3.6. Градиентный шаг

В общем случае градиентный шаг может зависеть от номера итерации, тогда мы будем писать не η, а η_k

- $\eta_k = c$ - это то что было у нас раньше - постоянный LR
- $\eta_k = \frac{1}{k}$ - здесь проблема в том, что мы можем на первом шаге стоять очень близко от глобального минимума и пролететь его
- $\eta_k = \lambda \left(\frac{s}{s_0+k}\right)^p$, λ, s_0, p - параметры

3.7. AdaGrad (Adaptive Gradient)

Сумма квадратов обновлений

$$g_{k-1,j} = (\nabla Q(\omega^{(k-1)}))^2$$

Формула метода AdaGrad:

$$G_{k,j} = G_{k-1,j} + g_{k-1,j} = G_{k-1,j} + (\nabla Q(\omega^{(k-1)}))^2$$

$$\omega_j^{(k)} = \omega_j^{(k-1)} - \frac{\eta}{\sqrt{G_{k,j} + \varepsilon}} \cdot (\nabla Q(\omega^{(k-1)}))_j$$

Этот метод использует адаптивный шаг обучения по каждой из координат веса - тем самым мы регулируем скорость сходимости метода.

Плюсы метода: происходит затухание величины шага

Минусы метода: $G_{k,j}$ монотонно возрастает, поэтому шаги укорачиваются и мы можем не успеть дойти до минимума

3.8. RMSPROP (Root Mean Square Propagation)

Метод реализует экспоненциальное затухание градиентов

Формулы метода RMSprop (усреднённый по истории квадрат градиента):

$$G_{k,j} = \alpha \cdot G_{k-1,j} + (1-\alpha) \cdot g_{k-1,j}$$

$$w_j^{(k)} = \omega_j^{k-1} - \frac{\eta}{\sqrt{G_{k,j} + \varepsilon}} \cdot (\nabla Q(\omega^{(k-1)}))_j$$

Если мы быстро сдвигались на последних шагах - то следующие будут маленькие

Если мы сдвигались медленно - то шаги будут большие

3.9. Практикум на питоне

3.9.1. Генерируем данные

```
import numpy as np
from matplotlib import pylab as plt
%pylab inline

from sklearn.datasets import make_regression


X, y = make_regression(n_samples=10000)
print(X.shape, y.shape)
```

3.9.2. Библиотечное решение

У нас 10000 объектов и 100 признаков. Для начала решим задачу аналитически "из коробки".

Решим сначала аналитически с помощью LinearRegression

```
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error

lr = LinearRegression()

lr.fit(X, y)

print(mean_squared_error(y, lr.predict(X)))
```

Посмотрим на свободный член модели (intercept_) и веса (coef_)

```
print(lr.intercept_, lr.coef_[:5])
```

Смотрим всего 5 весов, так как в реале их там 100.
Теперь решим с помощью градиентного спуска:

```
from sklearn.linear_model import SGDRegressor

sgd = SGDRegressor(alpha=0.00000001)
sgd.fit(X, y)

print(mean_squared_error(y, sgd.predict(X)))
```

И теперь точно также посмотрим свободный член и веса

```
print(sgd.intercept_, sgd.coef_[:5])
```