Национальный исследовательский университет ИТМО

Факультет информационных технологий и программирования

Прикладная математика и информатика

Методы оптимизации

Отчёт по лабораторной работе №3

Работу выполнили:

Ивченков Д. А., M32341

Султанов М. М., M32341

Трещёв А. С., M32341

Преподаватель:

Ким С. Е.

Санкт-Петербург

2023

Цель работы: изучение и исследование ньютоновских и квазиньютоновских методов, сравнение их с другими методами оптимизации.

Задачи:

1. Реализовать методы Gauss-Newton и Powell Dog Leg для решения нелинейной регрессии.
2. Реализовать метод BFGS.
3. Реализовать метод L-BFGS
4. Исследовать сходимость методов и сравнить их с методами, реализованными в предыдущих работах.

Использованные библиотеки:

- Numpy
- Matplotlib.pyplot

Gauss-Newton и Powell Dogleg

Методы Gauss-Newton и Powell Dog Leg являются итерационными методами нахождения минимума функции, которые могут быть применены для решения задач оптимизации. Оба метода могут использоваться для решения задач нелинейного наименьших квадратов (ННК), но Powell Dog Leg может быть применен и к задачам общей нелинейной оптимизации.

Метод Gauss-Newton основан на линеаризации функции, итеративном решении системы уравнений методом наименьших квадратов и оптимизации квадратичной функции. Этот метод сходится быстро, если начальное приближение близко к оптимальному решению. Однако он может столкнуться с проблемой неустойчивости, если матрица Якоби не положительно определена.

```
import numpy as np

def residual(params, x, y_meas):
    a, b, c = params
    y_pred = a * np.exp(-b * x) + c
    return y_pred - y_meas

def gauss_newton(residual, x, y_meas, params, maxiter=50, eps=1e-6):
    """
    Gauss-Newton метод оптимизации
    """
    for i in range(maxiter):
        r = residual(params, x, y_meas)
        J = approx_fprime(params, residual, epsilon=eps)
        p = np.linalg.lstsq(J, -r, rcond=None)[0]
        params += p
        if np.sum(np.abs(p)) < eps:
            break
    return params

def powell_dogleg(residual, x, y_meas, params, maxiter=50, eps=1e-6):
    """
    Powell Dog Leg метод оптимизации
    """
    x0 = params
    for i in range(maxiter):
        r = residual(x0, x, y_meas)
        J = approx_fprime(x0, residual, epsilon=eps)
        g = J.T @ r
        B = J.T @ J
        p_h = -g @ g / (g @ B @ g) * g
        p_gn = np.linalg.lstsq(B, -g, rcond=None)[0]
        if np.linalg.norm(p_gn) <= 2 * np.linalg.norm(p_h):
            p = p_gn
            alpha = 1.0
        else:
            p = p_h + (np.linalg.norm(p_h) ** 2 - np.linalg.norm(p_gn) ** 2) /
(2 * (p_h @ (p_gn - p_h)))
            alpha = np.linalg.norm(p_h) / (np.linalg.norm(p_h - p))
        x1 = x0 + alpha * p
        if np.sum(np.abs(x1 - x0)) < eps:
            break
        x0 = x1
    return x1
```

BFGS

Алгоритм BFGS (Broyden – Fletcher – Goldfarb – Shanno) – один из наиболее широко применяемых квазиньютоновских методов для нахождения минимума любых дважды дифференцируемых функций. Вместо точного вычисления гессиана функции считается приближённая оценка его обратной матрицы $H_k = B_k^{-1}$. Это позволяет избежать трудоёмких операций вычисления гессиана функции и обращения матрицы. Таким образом, сложность вычислений уменьшается с кубической до квадратичной зависимости от размерности.

На $k+1$-ом шаге H_{k+1} должна удовлетворять условию

$$H_{k+1} y_k = s_k$$

где

$y_k = \nabla f_{k+1} - \nabla f$ – изменение градиента,

$s_k = x_{k+1} - x_k = \alpha_k p_k$ – изменение точки, т.е. шаг алгоритма,

p_k – направление шага,

α_k – размер шага.

Матрица для следующей итерации может быть найдена по следующему принципу

$$H_{k+1} = argmin \, \|H - H_k\|$$

$$H = H^T, H y_k = s_k$$

И вычислена по формуле

$$H_{k+1} = (I - \rho_k s_k y_k^T) H_k (I - \rho_k y_k s_k^T) + \rho_k s_k s_k^T$$

$$\rho_k = \frac{1}{y_k^T s_k}$$

Тогда направление шага выбирается как

$$p_k = -H_k \nabla f_k$$

Для корректной работы метода размер шага должен вычисляться при помощи одномерного поиска с условиями Вольфе с параметрами $c_1 = 10^{-4}$, $c_2 = 0.9$.

Необходимо также выбрать начальное приближение обратного гессиана H_0, от которого будет зависеть траектория. В простом случае можно взять за основу единичную матрицу. Чтобы увеличить эффективность алгоритма, делается один шаг вдоль градиента функции в начальной точке и в качестве первого приближения принимается следующая матрица

$$H_0 = \frac{y_0^T s_0}{y_0^T y_0} I$$

Ниже представлена реализация метода BFGS.

```
def bfgs(f, x_0, epochs):
    iter_cnt = 0
    n = len(x_0)
    grad = gradient(f, x_0)
    p = -grad
    s = ternary_search_wolfe(f, x_0, p) * p
    y = gradient(f, x_0 + s) - grad
    I = np.identity(n)
    H = np.dot(y, s) / np.dot(y, y) * I
    x = x_0
    points = np.zeros((epochs + 1, n))
    points[0] = x_0
    while np.linalg.norm(s) > 1e-6 and iter_cnt < epochs:
        iter_cnt += 1
        p = np.dot(-H, grad)
        s = ternary_search_wolfe(f, x, p) * p
        x += s
        next_grad = gradient(f, x)
        y = next_grad - grad
        rho = 1 / np.dot(y, s)
        H = np.dot(I - rho * np.outer(s, y), np.dot(H, I - rho * np.outer(y,
s))) + rho * np.outer(s, s)
        grad = next_grad
        points[iter_cnt] = x
    return points, iter_cnt
```

В сравнении с различными видами градиентного спуска метод BFGS показывает более быструю и точную сходимость. Кроме того, данный алгоритм успешно справляется с оптимизацией в том числе и функций, с которыми у методов градиентного спуска возникают сложности. Однако количество вычислений функции на одну итерацию и требования по памяти находятся в квадратичной зависимости от размерности пространства, следовательно, заметно увеличиваются относительно предыдущих методов и вызывают проблемы при больших размерностях.

L-BFGS (limited memory BFGS)

При взаимодействии с матрицами в методе BFGS нам хочется тратить меньше памяти на хранение самой матрицы H_k – обратной к матрице «гессиана» оптимизируемой функции f в точке x_k(текущее значение аргумента)

Поэтому вместо того чтобы хранить матрицу размера $n \times n$ (где n – размерность вектора x) данный метод предлагает хранить лишь несколько векторов которые неявно представляют аппроксимацию искомой матрицы. Из-за возникающем линейном, а не квадратном требовании памяти, метод L-BFGS хорошо подходит для решения задачи поиска оптимума с большим количеством переменных. То есть, вместо обратного «гессиана» метод поддерживает историю последних m изменений положения x и соответственно градиента

$\nabla f(x)$. Размер истории может быть небольшим (часто m не больше 10). Эта история используется для неявного выполнения операций требующих произведения матрицы H

Алгоритм начинает свою работу с начальной точки x_0 и на каждой итерации обновляет это значение более точной оценкой

Производные функции $g_k = \nabla f(x_k)$ используются не только в качестве определения направления спуска, но и для формирования оценки матрицы «гессиана»

Вообще говоря, метод L-BFGS имеет много похожих моментов с другими квази-Ньютоновскими методами, но принципиально отличается от них тем, как высчитывается матричное произведение. Существует множество опубликованных подходов, использующих историю обновлений для формирования этого произведения. Здесь мы приводим общий подход, так называемую "рекурсию с двумя циклами".

Обозначим за x_k – текущее значение аргумента функции, $g_k \equiv \nabla f(x_k)$ – значение градиента при текущем значении аргумента оптимизируемой функции. Также предположим, что мы храним последние m изменений значений: $s_k = x_{k+1} - x_k \quad y_k = g_{k+1} - g_k$

Определим $\rho_k = \frac{1}{y_k^T s_k}$ (здесь и далее подразумеваем что векторы записаны в столбец, а транспонированные в строчку) и H_k - начальное приближение обратного «гессиана» с которого начинается наша оценка на итерации k

Алгоритм основан на формуле пересчёта «гессиана» в методе BFGS

На шаге k определим последовательность вектором $q_{k-m}, \dots, q_k$ как $q_k \coloneqq g_k$, $q_i \coloneqq (I - \rho_i y_i s_i^T) q_{i+1}$, где I – единичная матрица. То есть достаточно посчитать коэффициент $\alpha_i = \rho_i s_i^T q_{i+1}$ и вычислить вектор $q_i = q_{i+1} - \alpha_i y_i$, также определим последовательность векторов $z_{k-m}, \dots, z_k$ как $z_i \coloneqq H_i q_i$, и здесь снова рекурсивно мы посчитаем вектора по следующему правилу: $z_{k-m} \coloneqq H_k q_{k-m}$ и определив коэффициент $\beta_i \coloneqq \rho_i y_i^T z_i$ получим $z_{i+1} = z_i + (\alpha_i - \beta_i) s_i$. Послед проделанных вычислений значение вектора -z_k и будет нашим текущим направлением спуска. Таким образом можем вычислить направление спуска следующим образом:

```
q = g
size = len(last_updates_s)
for i in range(size - 1, -1, -1):
    alpha[i] = last_updates_ro[i] * np.dot(last_updates_s[i], q)
    q = q - alpha[i] * last_updates_y[i]
gamma = np.dot(last_updates_s[size-1], last_updates_y[size-1]) /
np.dot(last_updates_y[size - 1],

last_updates_y[size - 1])
```

```
matrix_h = gamma * np.identity(dim)
z = np.dot(matrix_h, q)
for i in range(size):
    beta[i] = last_updates_ro[i] * np.dot(last_updates_y[i], z)
    z = z + last_updates_s[i] * (alpha[i] - beta[i])
```

Где last_updates_s, last_updates_y, last_updates_rho - соответственно последние m изменений векторов s, y и числа ρ. Обращаю внимание также на то, что несмотря на умножение gamma * np.identity(dim) операция требует линейное количество памяти, и выполняется за линейное время, не квадратное. Полная реализация метода L-BFGS представлена ниже:

```
def l_bfgs(f, start_point, epoch=1000,  m=10):
    # инициируем некоторые константы и начальные значения
    grad_eps = 1e-7
    x = start_point
    dim = len(start_point)
    g = calc.gradient(f, x)
    points = np.zeros((epoch + 1, dim))
    points[0] = start_point

    alpha = np.zeros(m)
    beta = np.zeros(m)

    # заводим дэк для хранения не более m изменений значений
    last_updates_s = collections.deque()
    last_updates_y = collections.deque()
    last_updates_ro = collections.deque()

    # инициируем стартовое смещение
    z = 0.018 * calc.gradient(f, x)
    count_epochs = 0
    for k in range(1, epoch+1):

        # получаем новое значение x и высчитываем на delta_x и delta_gradient
        x = x - z
        points[k] = x
        g_1 = calc.gradient(f, x)
        s = points[k] - points[k - 1]
        y = g_1 - g
        ro = 1 / (np.dot(y, s))
        g = g_1

        # если норма градиента меньше эпсилон, то стоп
        if np.linalg.norm(g) < grad_eps:
            print("Потребовалось итераций " + str(k))
            count_epochs = k
            break
        if k > m:
            # в дэке должно быть не больше m элементов
            last updates s.popleft()
            last_updates_y.popleft()
            last_updates_ro.popleft()

        # добавляем элементы в дэк
        last_updates_s.append(s)
        last_updates_y.append(y)
        last_updates_ro.append(ro)

        # вычисляем новое смещение
```

```
        q = g
        size = len(last_updates_s)
        for i in range(size - 1, -1, -1):
            alpha[i] = last_updates_ro[i] * np.dot(last_updates_s[i], q)
            q = q - alpha[i] * last_updates_y[i]
        gamma = np.dot(last_updates_s[size-1], last_updates_y[size-1]) /
np.dot(last_updates_y[size - 1],

last_updates_y[size - 1])
        matrix_h = gamma * np.identity(dim)
        z = np.dot(matrix_h, q)
        for i in range(size):
            beta[i] = last_updates_ro[i] * np.dot(last_updates_y[i], z)
            z = z + last_updates_s[i] * (alpha[i] - beta[i])
    return x, points[:count_epochs+1]
```

Сравнение методов

Результаты работы для $f(x, y) = 0.01x^2 + y^2$

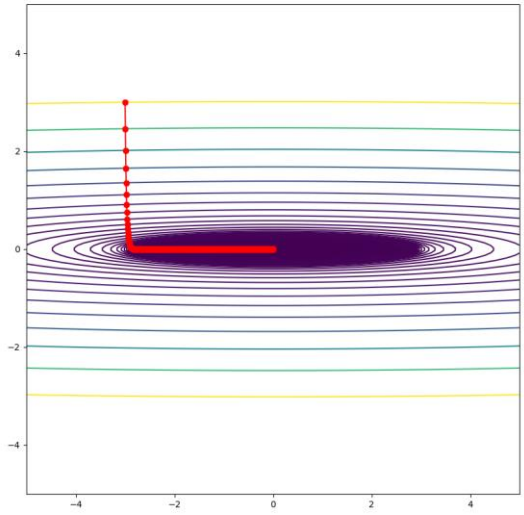


Рисунок 2 – Обычный градиентный спуск

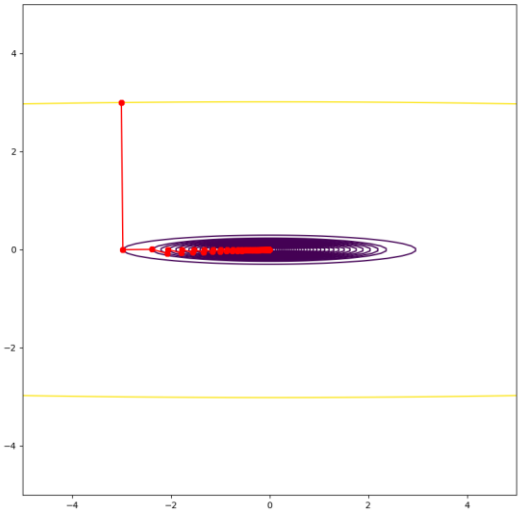


Рисунок 1 – Градиентный спуск с одномерным поиском

	Обычный GD	GD с одномерным поиском	Gauss-Newton	Powell Dogleg	BFGS	L-BFGS
Количество итераций	3552	132	25	37	34	5
Количество вычислений функции	14208	8122	3094	3523	2180	1180

Результаты работы для $f(x, y) = (1 - x)^2 + 100(y - x^2)^2$ (функция Розенброка)

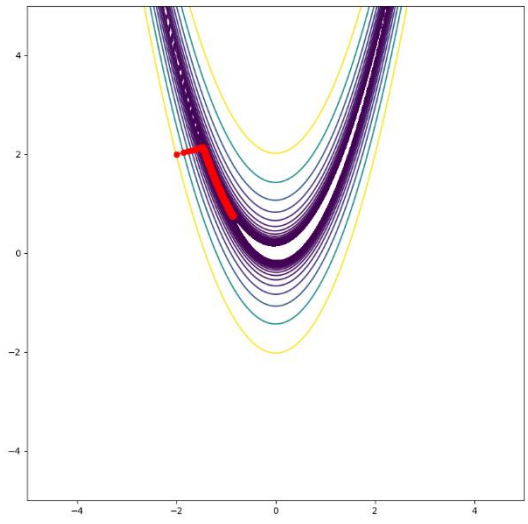
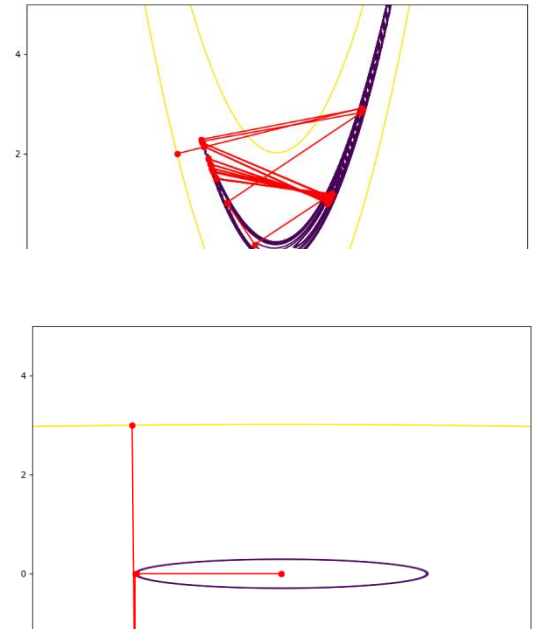


Рисунок 8 – Обычный градиентный спуск



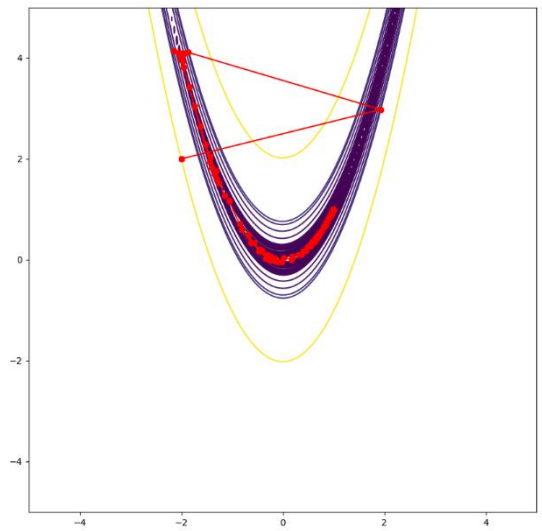


Рисунок 6 – BFGS

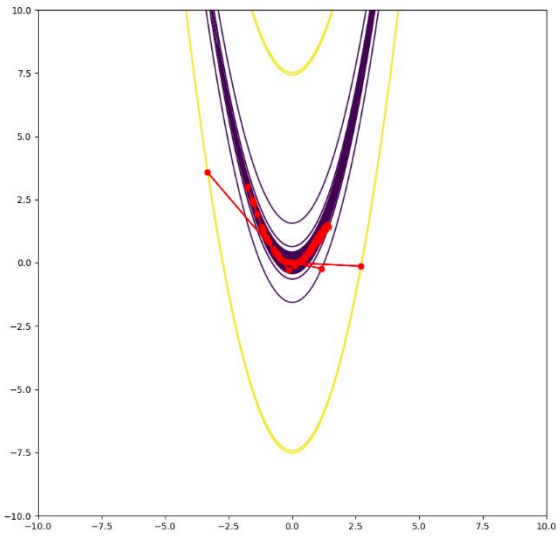


Рисунок 5 – L-BFGS

	Обычный GD	GD с одномерным поиском	Gauss-Newton	Powell Dogleg	BFGS	L-BFGS
Количество итераций	23159	209233	102	98	34	119
Количество вычислений функции	92636	12972384	122704	20013	11800	6712

Результаты для $f(x,y) = (1.5 - x + xy)^2 + (2.25 - x + xy^2)^2 + (2.625 - x + xy^3)^2$ (функция Била)

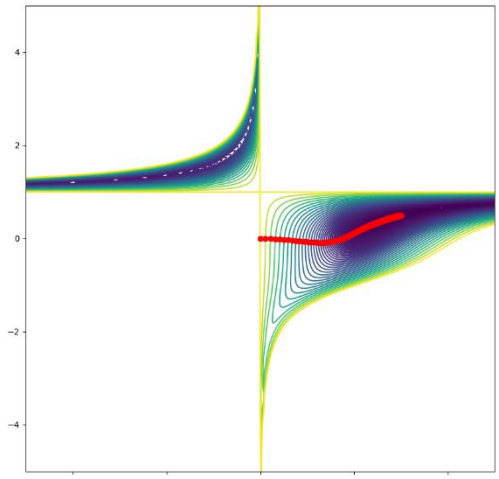


Рисунок 12 – Обычный градиентный спуск

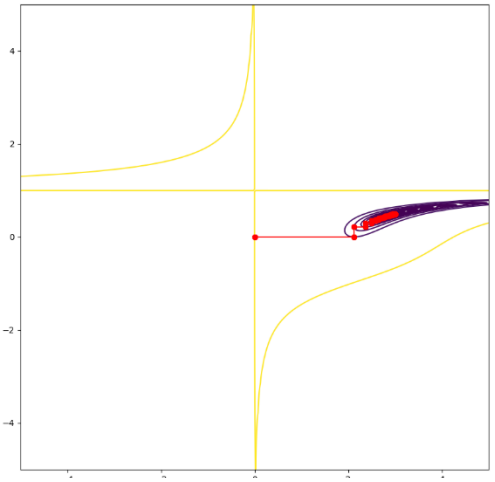


Рисунок 11 – Градиентный спуск с одномерным поиском

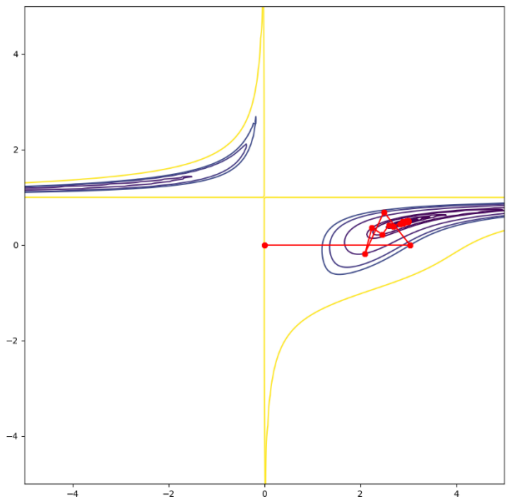


Рисунок 10 – BFGS

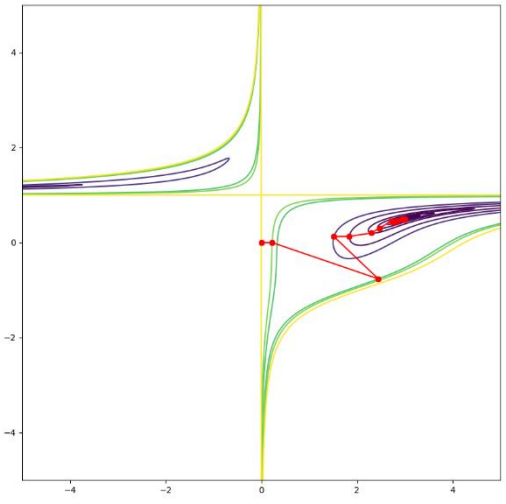


Рисунок 9 – L-BFGS

	Обычный GD	GD с одномерным поиском	Gauss-Newton	Powell Dogleg	BFGS	L-BFGS
Количество итераций	2592	628	92	67	26	16
Количество вычислений функции	10368	38874	110491	34619	9900	5043

Реализованные методы были сравнены по эффективности с реализованными модификациями стохастического градиентного спуска. Как видно, реализованные в данной работе методы сходятся заметно быстрее предыдущих методов, но производят больше вычислений.

Критерий сравнения	SGD	Momentum	Nesterov	Adagrad	RMSProp	Adam	Gauss-Newton	Powell Dogleg	BFGS	L-BFGS
Среднее количество итераций	8925	3280	5242	3067	3535	2129	235	189	41	128
Среднее количество вычислений	24950	49209	94368	46014	58739	66008	190541	79762	8325	16743
Среднее количество вычислений на итерацию	2,79	15	18	15	16,62	31	810,81	422,1	203	130