Национальный исследовательский университет ИТМО
Факультет информационных технологий и программирования
Прикладная математика и информатика

Методы оптимизации
Отчет по лабораторной работе №3

Работу выполнили:
Володько А.А., M32321
Ярунина К.А., M32371
Преподаватель:
Шохов М.Е.

Санкт-Петербург
2023

Постановка задачи:

1. Реализуйте методы Gauss-Newton и Powell Dog Leg для решения нелинейной регрессии. Сравнить эффективность с методами, реализованными в предыдущих работах.

2. Реализовать метод BFGS и исследовать его сходимость при минимизации различных функций. Сравнить с другими реализованными методами.

*. Реализовать и исследовать метод L-BFGS.

```
Nonlinear regression
Numerical differentiation
Gauss-Newton method
Trust-region method
Powell's Dog Leg method
BGFS method
L-BGFS method
Сравнение методов на линейной регрессии
Сравнение методов на полиномиальной регрессии
```

`Nonlinear regression`

Как и в случае линейной/полиномиальной регрессии, нам даны некоторые точки (x_i, y_i), предположительно принадлежащие некоторой функции, и наша задача – найти эту функцию. Однако если в случае линейной регрессии мы ограничивались функциями вида $y = w_1 x + w_2$, и искали подходящие коэффициенты w_i, то сейчас функция может принимать более сложный вид: например $y = w_1 \sin(w_2 x) + 2^{w_3 x} + w_4 x^4$.

Переходя к конкретной задаче, пусть нам дана функция $g(w, x)$ (например, $g(w, x) = w_1 \sin(w_2 x) + 2^{w_3 x} + w_4 x^4$) и набор точек (x_i, y_i). Тогда чтобы найти подходящие коэффициенты w_i, нам необходимо минимизировать функцию

$$f(w) = \frac{1}{2} \sum_{i=0}^{n} (g(w, x_i) - y_i)^2$$

Эту задачу называют `nonlinear least squares problem`, ее и будем решать.

Введем следующие обозначия: $r_i(w) = g(w, x_i) - y_i$, $r(w) = (r_0(w) \dots r_{n-1}(w))^T$.

В таком случае f можно переписать как:

$$f(w) = \frac{1}{2} \sum_{i=0}^{n} r_i(w)^2$$

Якобиан вектор-функции $r(w)$:

$$(J_r)_{ij} = \frac{\partial r_i}{\partial w_j}$$

Заметим, что

$$\nabla f(w) = J_r(w)^T r(w)$$

$$\nabla^2 f(w) = J_r(w)^T J_r(w) + \sum_{i=0}^{n} r_i(w) \nabla^2 r_i(w)$$

Интересный факт №1: 2 слагаемое $\equiv 0$ для линейной g.

Интересный факт №2: 2 слагаемое в целом вносит куда меньший вклад, чем первое.

`Numerical differentiation`

Раз уж мы посвятили почти целую лекцию численному дифференцированию (`numerical differentiation`), то будем им пользоваться. В следующих алгоритмах нам понадобится вычислять значения первой и второй частной производной, так что посмотрим как они работают.

В случае первой производной f все просто: разложим f в ряд Тейлора в точке $x + h$:

$$f(x + h) = f(x) + f'(x)h + O(h) \text{ , откуда получаем } f'(x) = \frac{f(x + h) - f(x)}{h} + O(h)$$

Чтобы снизить погрешность, можно сделать такой трюк:

$$\begin{cases} f(x + h) = f(x) + f'(x)h + O(h) \\ f(x - h) = f(x) - f'(x)h + O(h) \end{cases} -$$

$$f'(x) = \frac{f(x + h) - f(x - h)}{2h} + O(h^2)$$

Эта формула очевидно распроняется на случай частной производной функции от нескольких переменных:

$$\frac{\partial f}{\partial x_i}(x_1..x_n) = \frac{f(x_1..x_i + h..x_n) - f(x_1..x_i - h..x_n)}{2h} + O(h^2)$$

Для вторых производных по разным переменным тоже есть формулы, к сожалению они слишком больно выводятся, однако запишем, что пользоваться будем такой:

$$\frac{\partial^2 f}{\partial x \partial y}(x, y) \approx \frac{f(x + h, y + k) - f(x + h, y - k) - f(x - h, y + k) + f(x - h, y - k)}{4hk}$$

`Gauss-Newton method`

Рассмотрим обычный метод Ньютона: $\nabla^2 f(w)p = -\nabla f(w)$: p – искомое направление, которое мы ищем исходя из решения этой задачи. Но это вычислительно сложно, поэтому рассмотрим метод Гаусса-Ньютона, который делает следующее упрощение: из выведенного равенства для $\nabla^2 f(w)$ оставляем только первое слагаемое. Теперь нам нужно

решить такую задачу: $J_r(w)^T J_r(w)p = -J_r(w)^T r(w)$.
Тогда следующее приближение w мы находим как

$$w_{k+1} = w_k - J_r(w)^+ r(w_k)$$

где $J_r(w)^+$ – псевдообратная матрица к $J_r(w)$.
Реализация:

```
def get_polynomial_reg(dim):
    return lambda w, x: sum([w[i]*x**i for i in range(dim)])

def get_r(f, x: np.array, y: np.array):
    return lambda w: np.array([(f(w, x[index])-y[index]) for index in range(x.size)])

def get_jacobian(f, x: np.array):
    return lambda w: np.array([
        np.array([get_derivative(f, w, j, x[i]) for j in range(w.size)])
        for i in range(x.size)])

def gauss_newton(
        w: np.array, x: np.array, y: np.array,
        reg=None,
        brk=1e-5, epoch=300):
    if reg is None:
        reg = get_polynomial_reg(w.size)
    r = get_r(reg, x, y)
    jacobian = get_jacobian(reg, x)
    for ep in range(epoch):
        step = np.linalg.pinv(jacobian(w)) @ r(w)
        if np.linalg.norm(step) < brk:
            return w, ep
        w = w-step
    return w, epoch
```

`Trust-region method` – метод доверительного региона
Идея заключается в том, чтобы построить некую модель функции, определить область применимости модели и минимизировать модель, а не саму функцию. Это полезно, если сама функция сложновычислима, модель позволяет упростить вычисления.
Модель чаще всего используется квадратичная:

$$m_k(p) = f_k + g_k^T p + \frac{1}{2} p^T B_k p$$

если в качестве f_k взять текущее значение функции, g_k градиент, и B_k гессиан, то мы получим квадратичное приближение функции. Задача ставится так: мы хотим найти $\min m_k(p) : ||p|| \leq \Delta_k$.
Введем следующий коэффициент:

$$\rho_k = \frac{f(x_k) - f(x_k + p_k)}{m_k(0) - m_k(p)}$$

отношение приращения функции к приращению модели на k-ом шаге. Если модель уменьшилась значительно сильнее, чем функция, значит модель далека от функции, и регион нужно уменьшить, если же соответствие хорошее, то регион можно увеличить.

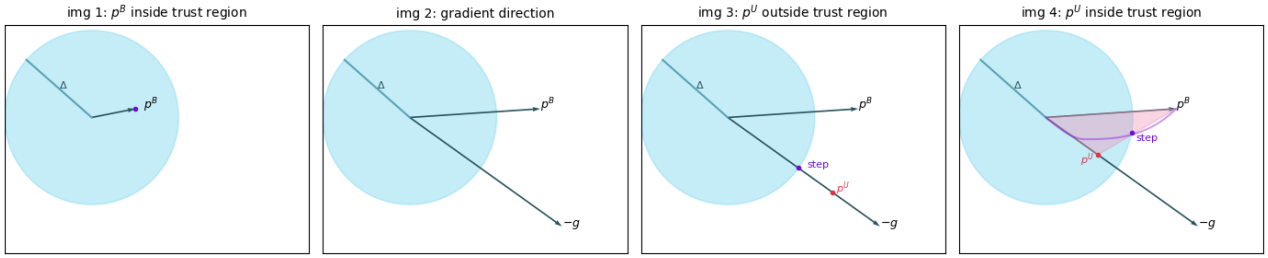
`Powell's Dog Leg method`
Возвращаемся к задаче – $\min m_k(p) : ||p|| \leq \Delta_k$. Мы можем, например, решить ее игнорируя ограничение, и если решение удовлетворяет ограничению, то мы получили ответ. Второй вариант – упростить модель до линейной:

$$m_k(p) = f_k + g_k^T p, ||p|| \leq \Delta_k$$

тогда без ограничения мы бы делали просто градиентный спуск. Идея Dog Leg метода в том, чтобы объединить эти подходы.
На каждой итерации смотрим на $p^B = -B^{-1}g$ – решение квадратичной модели, если p^B внутри trust-region, то можем его взять `[img 1]`. Иначе, смотрим на направление антиградиента: вдоль этого вектора найдем минимум модели: $\min m(-\tau g)$, $|| - \tau g|| \leq \Delta$ `[img 2]`.

$$p^U = -\frac{g^T g}{g^T B g} g$$

– минимум квадратичной модели в направлении антиградиента. Если p^U вне trust region, то можно просто взять точку на границе и шагнуть туда `[img 3]`, иначе строим линию от p^U к p^B – минимум будет двигаться по дуге, ограниченной полученным треугольником, там и будем его искать `[img 4]`.



Собственно, в данном случае нам достаточно просто взять в качестве минимума точку пересечения отрезка, соединяющего p^U и p^B, и радиуса trust-region.
Как: введем функцию

$$\widetilde{p}(\tau) = p^U + \tau(p^B - p^U),\ \tau \in (0, 1)$$

Чтобы найти τ, соответствующий `step` (на `[img 4]`), нужно решить уравнение

$$|\widetilde{p}(\tau)|^2 = \Delta^2$$

$$|p^U + \tau(p^B - p^U)|^2 = \Delta^2$$

$$(p^U)^2 + 2p^U\tau(p^B - p^U) + \tau^2(p^B - p^U)^2 = \Delta^2$$

$$(p^B - p^U)^2\tau^2 + 2p^U(p^B - p^U)\tau + (p^U)^2 - \Delta^2 = 0$$

$$\tau = \frac{-2p^U(p^B - p^U) + \sqrt{4(p^U)^2(p^B - p^U)^2 - 4(p^B - p^U)^2((p^U)^2 - \Delta^2)}}{2(p^B - p^U)^2}$$

$$\tau = \frac{-2p^U(p^B - p^U) + 2(p^B - p^U)\Delta}{2(p^B - p^U)^2} = -\frac{(p^U - \Delta)(p^B - p^U)}{(p^B - p^U)^2}$$

Тогда искомый шаг можно найти как

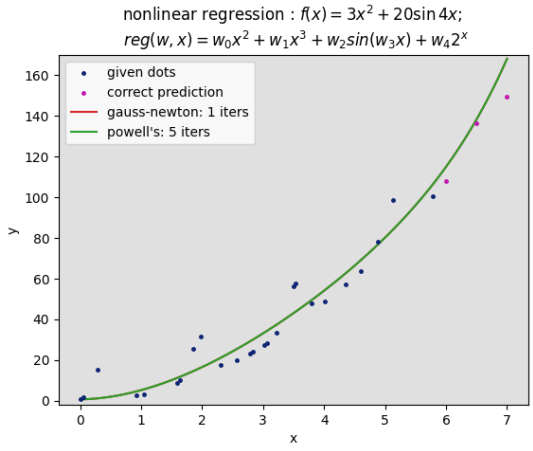
$$p = p^U + \tau(p^B - p^U)$$

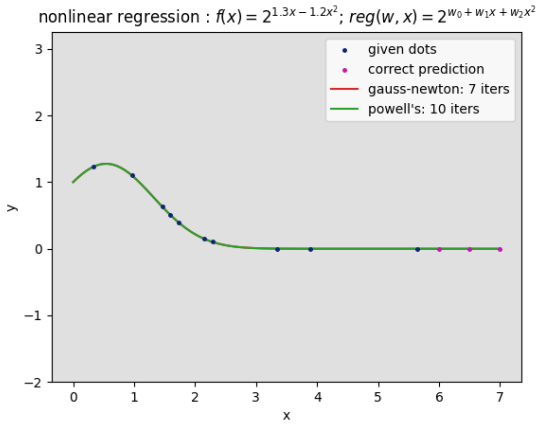
Реализация:

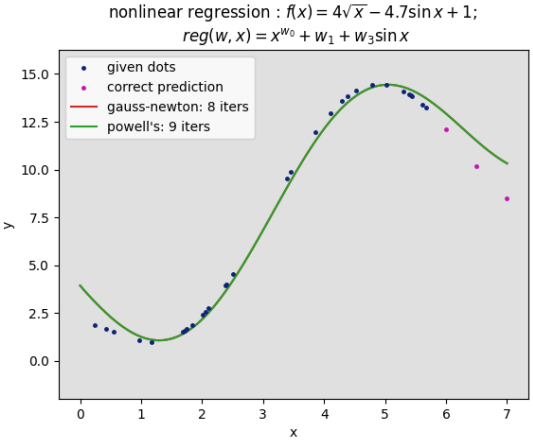
```
def get_loss_f(f, x: np.array, y: np.array):
    return lambda w: 0.5 * sum([(f(w, x[i])-y[i])**2 for i in range(x.size)])

def get_jac(f, dim: int):
    return lambda w: np.array([get_derivative(f, w, i) for i in range(dim)])

def get_hes(f, dim: int):
    def hes(w):
        H = np.zeros((dim, dim), dtype=float)
        for i in range(dim):
            for j in range(i+1, dim):
                H[i][j] = get_second_derivative(f, w, i, j)
        H = H + H.T
        for i in range(dim):
            H[i][i] = get_second_derivative(f, w, i, i)
        return H
    return hes

def powell_dog_leg(
        w: np.array, x: np.array, y: np.array,
        reg=None,
        trust_region=1.0, max_trust_region=100.0, nu=0.15,
        brk=1e-3, epoch=300):
    if reg is None:
        reg = get_polynomial_reg(w.size)
    loss_f = get_loss_f(reg, x, y)
    jacobian, hessian = get_jac(loss_f, w.size), get_hes(loss_f, w.size)
    for ep in range(epoch):
        g, b = jacobian(w), hessian(w)
```

```
        pb = -np.linalg.pinv(b) @ g
        if np.linalg.norm(pb) <= trust_region:
            p = pb
        else:
            pu = - ((g @ g) / (g @ (b @ g))) * g
            norm_pu = np.linalg.norm(pu)
            if norm_pu >= trust_region:
                p = trust_region * pu / norm_pu
            else:
                diff = pb-pu
                tau = - ((pu-trust_region) @ diff)/(diff @ diff)
                p = pu+tau*diff
        model_red = -(g @ p + 0.5 * p @ (b @ p))
        rho = 1e99 if model_red == 0.0 else (loss_f(w) - loss_f(w + p)) / model_red
        norm_p = np.linalg.norm(p)
        if rho < 0.25:
            trust_region = 0.25 * norm_p
        elif rho > 0.75 and norm_p == trust_region:
            trust_region = min(2.0 * trust_region, max_trust_region)
        if rho > nu:
            w = w + p
        if np.linalg.norm(g) < brk:
            return w, ep
    return w, epoch
```

Посмотрим на работу вышеописанных алгоритмов `Gauss-Newton` и `Powell's Dog Leg` на примере следующих нелинейных регрессий:







results	f1	f2	f3
gauss-newton	[5.54211, -0.476, 0.0, 0.0, 0.20476]	[0.0, 1.3, -1.2]	[1.14396, 3.93517, -4.37503]
dog leg	[5.54211, -0.476, 0.0, 0.0, 0.20476]	[0.0, 1.3, -1.2]	[1.14396, 3.93518, -4.37504]

Видим, что справляются они на этих примерах хорошо, их результаты идентичны с точностью до погрешности, однако `Powell's` метод требует больше итераций. Это вызвано как раз тем, что второй метод использует `trust-region`

подход, что влечет во-первых пропуск некоторых итераций (в реализации видно, что мы приближаем точку только в случае `rho>nu`, иначе просто пропускаем), а во-вторых в целом аппроксимируя мы теряем точность шага в пользу меньшего количества вычислений.

`BGFS method`

До этого мы рассматривали так называемые ньютоновские методы, их основная проблема в том, что необходимо напрямую вычислять гессиан функции, это, к сожаленкию, бывает вычислительно дорого. Собственно, метод `BFGS`, относящийся к классу квазиньютоновских, направлен на то, чтобы аппроксимировать гессиан, используя информацию из предыдущих итераций: на каждой итерации алгоритма матрица Гессе обновляется с использованием последних значений градиента в текущей и предыдущей точках, что позволяет алгоритму быстрее адаптироваться к изменению формы функции, чем другим методам, которые используют только информацию из предыдущей точки. Кроме того, апроксимация матрицы Гессе с учетом предыдущих итераций значений позволяет алгоритму достаточно хорошо работать на функциях с большим количеством переменных, поскольку не требуется вычислять матрицу целиком на каждой итерации. Стоит также отметить, что при увеличении числа переменных в функции увеличивается вероятность, что матрица Гессе может быть быть плохо обусловлена на какой-то итерации, то есть собственные числа матрицы становятся близкими к нулю, но за счет использования апроксимации, алгоритм остается достаочно эффективен даже в этом случае.

Здесь мы будем минимизировать обычную функцию $f(w)$, без каких либо ограничений (кроме, очевидно, C^2). Инициализируем приближение обратного гессиана H^{-1} обычной единичной матрицей (нам просто необходима невырожденная матрица с низким числом обусловленности, единичная идеально подходит).

Направление градиентного спуска в таком случае определяется как

$$p = -H^{-1} \cdot \nabla f(w)$$

а коэффициент α перед ним находится используя линейный поиск (здесь используем поиск с учетом условий Вольфе из 1 лабораторной). Тогда следующее приближение w вычисляется как

$$w_{j+1} = w_j + \alpha_j p_j$$

Обратно к гессиану: чтобы сделать следующее приближение нам понадобится ввести следующие обозначения:

$$s_j = w_j - w_{j-1}$$

$$y_j = \nabla f(w_j) - \nabla f(w_{j-1})$$

$$\rho_j = \frac{1}{y^T s}$$

и обратный гессиан обновляется как

$$H_{j+1}^{-1} = (I - \rho_j s_j y_j^T) H_j^{-1} (I - \rho_j y_j s_j^T) + \rho_j s_j s_j^T$$

Реализация:

```python
def check_wolfe(f, df, x, grad, alpha, c1=1e-5, c2=0.9):
    if f(x + alpha * -grad) > f(x) - c1 * alpha * grad @ grad:
        return False
    if grad @ df(x + alpha * -grad) > c2 * grad @ grad:
        return False
    return True

def bfgs(
        f, w: np.array,
        brk=1e-5, epoch=100):
    identity_matrix, grad_f = np.eye(w.size), get_gradient(f)
    hi = identity_matrix  # approximate hessian inverse
    g = grad_f(w)
    for ep in range(epoch):
        p = -hi @ g
        alpha = 1
        while not check_wolfe(f, grad_f, w, g, alpha):
            alpha /= 2
            if alpha == 0:
                alpha = 1e-10
                break
        w_prev, g_prev = w, g
        w = w + alpha * p
```

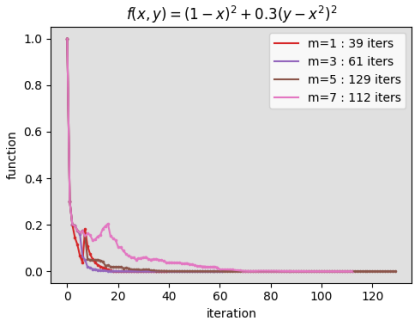
```
        g = grad_f(w)
        s, y = w - w_prev, g - g_prev
        if np.linalg.norm(s) < brk:
            return w, ep
        rho = 1.0 / (y @ s)
        hi = (identity_matrix - rho * np.outer(s, y)) \
            @ hi \
            @ (identity_matrix - rho * np.outer(y, s)) \
            + rho * np.outer(s, s)
    return w, epoch
```

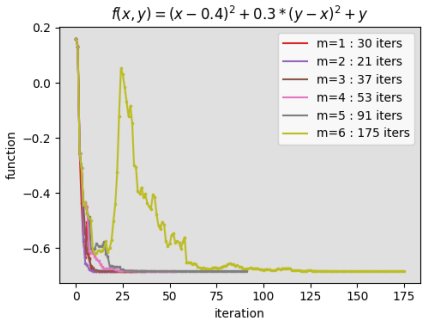
L-BGFS method

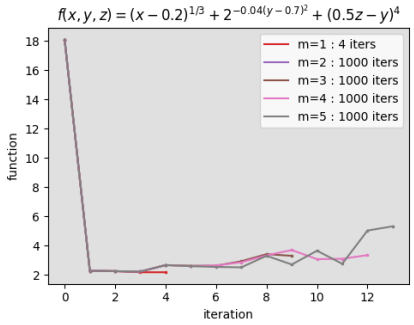
L-BGFS метод также относится к классу квазиньютоновских, то есть не вычисляет гессиан функции напрямую. Он аппроксимирует алгоритм BGFS, используя ограниченную память для хранения информации о предыдущих итерациях. Благодаря требованию линейной памяти, метод L-BGFS особенно хорошо подходит для оптимизационных проблем с большим количеством переменных, не требуя больших объемов памяти.
BGFS хранил полное приближение обратного гессиана размерности $n \times n$ (где n - количество переменных в функции), L-BGFS хранит только несколько векторов (приращения x и $\nabla f(x)$ на последних m итерациях), которые представляют собой неявное приближение. Обычно размер истории m используют небольшим (часто меньше 10).
Реализация:

```
def lbfgs(
        f, w: np.array,
        m=3,
        brk=1e-5, epoch=300):
    identity_matrix, grad_f = np.eye(w.size), get_gradient(f)
    g = grad_f(w)
    d = -g
    s_y_rho_prev = []  # queue of last m values of s,y,rho
    for ep in range(epoch):
        x_prev, g_prev = w, g
        alpha = 1
        while not check_wolfe(f, grad_f, w, g, alpha):
            alpha /= 2
            if alpha == 0:
                alpha = 1e-10
                break
        w = w + alpha * d
        g = grad_f(w)
        s, y = w - x_prev, g - g_prev
        ys = y @ s
        if abs(ys) == 0.0:
            continue
        rho = 1.0 / ys
        s_y_rho_prev.append([s, y, rho])
        if ep >= m:
            s_y_rho_prev.pop(0)  # support queue size=m
        if np.linalg.norm(s) < brk:
            return w, ep
        q = g
        gammas = []
        q_size = len(s_y_rho_prev)
        for k in range(q_size-1, -1, -1):
            s_k, y_k, rho_k = s_y_rho_prev[k]
            gamma = rho_k * s_k @ q
            gammas.insert(0, gamma)
            q = q - gamma * y_k
        r = q * (s @ y) / (y @ y)
        for k in range(q_size-1, -1, -1):
            s_k, y_k, rho_k = s_y_rho_prev[k]
            b = rho_k * y_k @ r
            r = r + s_k * (gammas[k] - b)
        d = -r
    return w, epoch
```

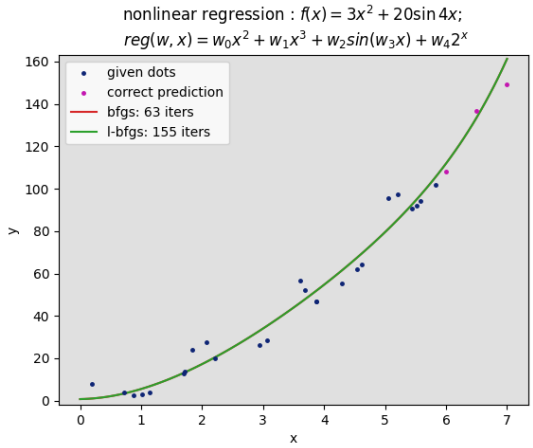
Посмотрим на работу метода на примере следующих нелинейных функций в зависимости от выбора параметра m:

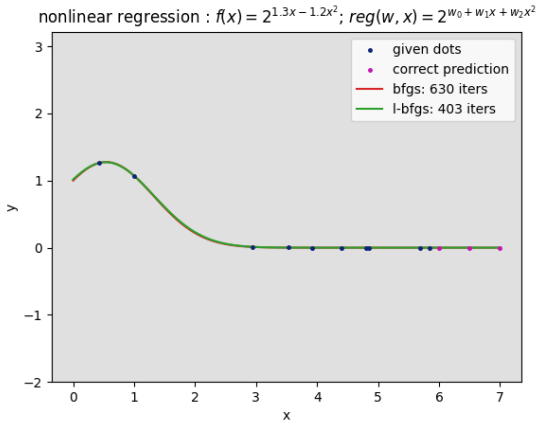


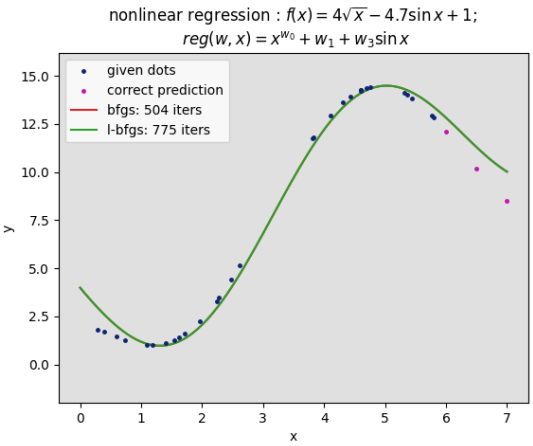




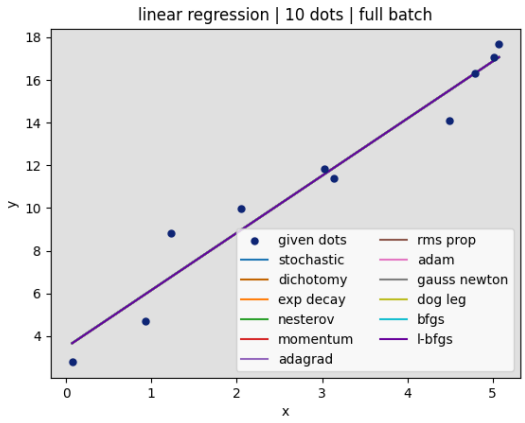
На графиках представлены зависимости значений минимизируемых функций от итераций. Мы видим, что на данных примерах оптимальное значение `m` около 1-2, это связано с тем, что зачастую в начальных итерациях приращения аргумента и градиента функции далеки от значений в последующих, то есть если алгоритм на первой же итерации сильно приблизился к минимуму, (как это здесь зачастую и происходит),а дальше идет намного более мелкими шагами, то учитывание первой итерации может вызвать проблемы со сходимостью.
Ниже представлены графики-примеры работы алгоритмов `BFGS` и `L-BFGS` на некоторых нелинейных регрессиях, мы видим, что `BFGS` тратит в среднем меньше итераций, и связано это с тем, что там мы все таки используем аппроксимированный гессиан в явном виде.

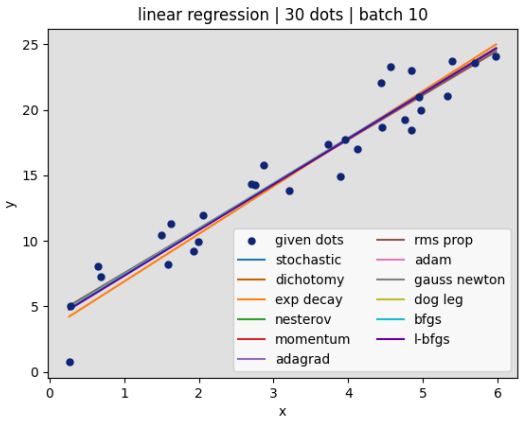






`Сравнение методов на линейной регрессии`
Рассмотрим задачу нахождения линейной регрессии на 10 и 30 начальных точках (для 10 мы использовали полный батч на методах, реализующих `minibatch` версии, на 30 `batch=10`).
В каждом случае запуски были проведены на 5 разных начальных точках со значениями мо модулю до 20 по каждой координате, и на графиках изображены средние результаты работы каждого из методов.





linear regression | 10 dots | full batch

method	mean res	mean loss	var loss	mean iter	var iter	mean time	var time	mean mem	var mem
stochastic	[3.4807, 2.67737]	5.1775	0.0	455	1612	0.0501	0.0	3852	1982709
dichotomy	[3.48072, 2.67736]	5.1775	0.0	150	4831	0.248	0.0128	2624	200704
exp decay	[3.47506, 2.6788]	5.1777	0.0	435	6728	0.0476	0.0001	2572	5570
nesterov	[3.48144, 2.67717]	5.1775	0.0	137	178	0.0156	0.0	2585	573
momentum	[3.48074, 2.67736]	5.1775	0.0	217	661	0.0244	0.0	2560	409
adagrad	[3.48077, 2.67735]	5.1775	0.0	195	4336	0.0222	0.0001	2598	573
rms prop	[3.48079, 2.67734]	5.1775	0.0	432	45050	0.05	0.0006	2656	6553
adam	[3.48083, 2.67733]	5.1775	0.0	240	889	0.0293	0.0	3494	27197
gauss newton	[3.48083, 2.67733]	5.1775	0.0	1	0	0.0009	0.0	4684	1386741
dog leg	[3.48083, 2.67733]	5.1775	0.0	4	5	0.0077	0.0	5856	503398
bfgs	[3.48057, 2.67743]	5.1776	0.0	595	1745	0.3429	0.0005	3673	393379
l-bfgs	[3.47707, 2.67839]	5.1776	0.0	1110	39887	0.6519	0.0138	3916	245

linear regression | 30 dots | batch 10

method	mean res	mean loss	var loss	mean iter	var iter	mean time	var time	mean mem	var mem
stochastic	[3.97574, 3.46684]	41.4351	0.0	2000	0	0.2258	0.0003	2521	163
dichotomy	[3.96698, 3.46605]	41.433	0.0	178	8149	0.8289	0.1929	2272	6144
exp decay	[3.26651, 3.63496]	46.0927	64.6866	936	6009	0.0996	0.0001	2566	163
nesterov	[3.99072, 3.4555]	41.4423	0.0	2000	0	0.221	0.0001	3116	245
momentum	[3.9693, 3.46499]	41.4332	0.0	2000	0	0.2181	0.0	2579	245
adagrad	[3.99938, 3.46215]	41.4848	0.0018	2000	0	0.2177	0.0	3123	245
rms prop	[4.14446, 3.3982]	42.8844	0.0989	2000	0	0.2247	0.0	3136	0
adam	[3.97396, 3.46395]	41.4359	0.0	2000	0	0.2341	0.0	3392	0
gauss newton	[3.96713, 3.46602]	41.433	0.0	1	0	0.0017	0.0	4128	0
dog leg	[3.96713, 3.46602]	41.433	0.0	4	1	0.0211	0.0	5344	325632
bfgs	[3.96454, 3.46642]	41.4341	0.0	1946	3219	3.0929	0.0146	3347	245
l-bfgs	[3.84896, 3.48643]	43.6812	5.6596	2000	0	3.2139	0.0004	3584	0

В целом графики выглядят многообещающе, значения на методах практически не отличаются друг от друга и выглядят правдоподобно (мы подбирали все параметры так, чтобы результаты были наиболее близки к искомым). Посмотрим на данные таблицы.

Здесь представлены `mean res` – среднее значение полученного argmin, `mean` и `var loss`, среднее значение и дисперсия функции потерь в полученной точке (эти параметры, опять же, близки друг к другу за счет подбора других параметров методов, таких как начальное значение `learning rate`, `gamma`, `decay rate` в случае экспоненциального затухания).

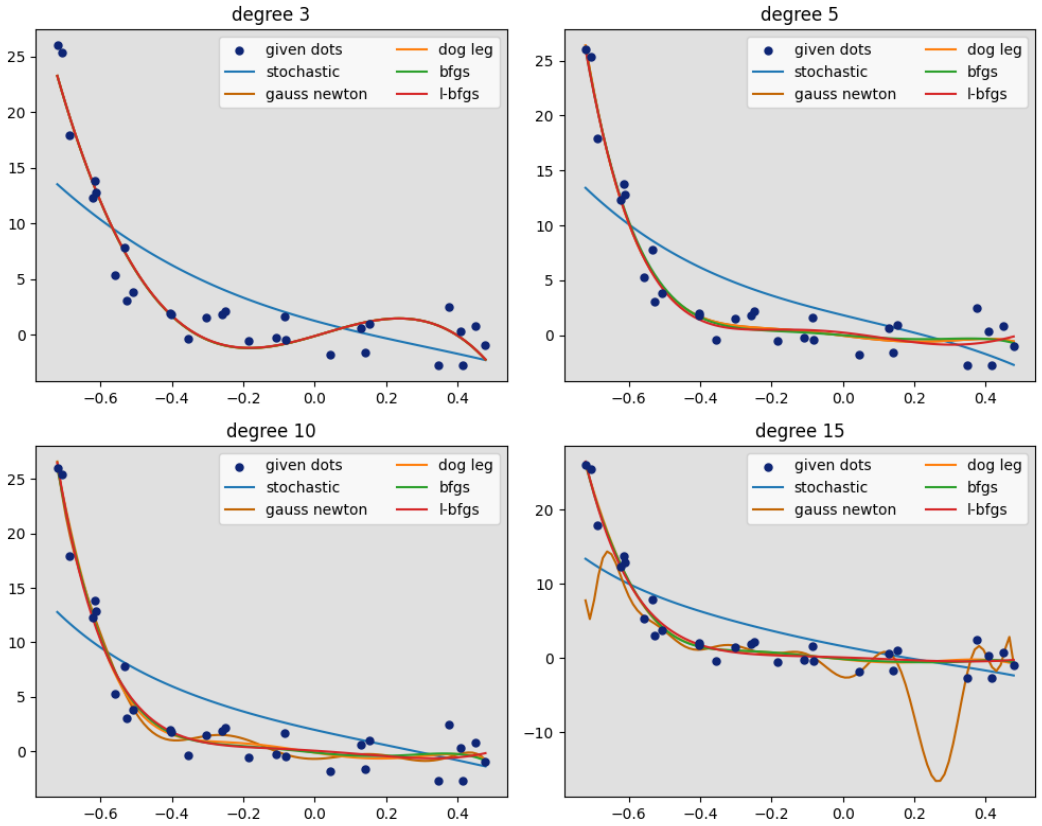
`mean` и `var iter` – средние значения и дисперсия затраченных каждым алгоритмом итераций. Первое, что бросается в глаза – итерации `Gauss-Newton` и `Powell's Dog Leg` методов. Возвращаясь к описанию нелинейной регрессии, где

мы говорили о том, что в случае линейной регрессии 2 слагаемое уравнении

$$\nabla^2 f(w) = J_r(w)^T J_r(w) + \sum_{i=0}^{n} r_i(w) \nabla^2 r_i(w)$$

(которое мы как раз в этих методах игнорируем) действительно $= 0$, соответственно наш шаг на каждой итерации фактически не теряет в точности, что влечет минимальное количество затраченных итераций. В случае алгоритмов `BFGS` и `L-BFGS`, намного больше итераций в среднем, чем в том числе методы из предыдущих лабораторных. С чем это связано: заметим, что для задачи линейной (и полиномиальной, кстати, тоже) регрессии минимизируемая функция $f(w)$ всегда квадратична, а значит, гессиан константный (от w не зависит), так как состоит из вторых производных. Если посмотреть на изменения гессиана в этих методах на линейной регрессии, можно в этом убедится – аппрокимация гессиана действительно будет константной начиная с 1 же итерации. Это значит, что в случае полиномиальной регрессии мы делаем фактически обычный градиентный спуск, однако вместо `learning rate` используем аппрокимацию гессиана, что может быть менее оптимально, чем даже взятие в качестве коэффициента константы (мы видим, что у стохастического градиентного спуска, соответсвующего обычному спуску с постоянным шагом на `batch=10`, итераций в среднем меньше).
Говоря о времени работы, учитывая, что оно в любом случае пропорциональны затраченным итерациям, явно видно, что алгоритмы `BFGS` и `L-BFGS` работают наиболее долго, в том числе по вышеописанным причинам.
Стоит отметить, что согласно использованному способу измерения затраченной памяти, (а именно `tracemalloc`), у `Gauss-Newton` и `Powell's Dog Leg` показатели наиболее высокие – сказывается необходимость хранения и вычисления гессиана – матрицы $n \times n$. У `BFGS` памяти затрачено меньше, вероятно из-за особенностей реализации, в остальных же случаях гессиан мы не используем, и поэтому память в целом итерациям пропорциональна.

`Сравнение методов на полиномиальной регрессии`



polynomial regression | 30 dots									
deg	method	mean loss	var loss	mean iter	var iter	mean time	var time	mean mem	var mem
4	stochastic	304.4277	1666.1105	300	0	0.0885	0.0	3737	2108784
	gauss newton	66.1632	0.0	1	0	0.0037	0.0	4947	1165148
	dog leg	66.1632	0.0	9	0	0.1613	0.0001	6579	10895
	bfgs	66.1632	0.0	258	1292	1.8043	0.7535	3680	58163
	l-bfgs	66.1939	0.0038	300	0	1.5654	0.002	3654	163
6	stochastic	320.0759	6123.4531	300	0	0.1716	0.0	2515	655
	gauss newton	34.4962	0.0	1	0	0.0067	0.0	4044	655
	dog leg	34.4962	0.0	16	1	0.7026	0.0027	5497	22282
	bfgs	36.2299	12.0228	267	4199	3.1152	0.9812	3398	163
	l-bfgs	35.9238	0.6328	300	0	2.6268	0.0004	3648	0
11	stochastic	323.7787	5871.3828	300	0	0.5499	0.0022	2515	655
	gauss newton	31.5117	0.0	300	0	2.7373	0.0002	5056	715571
	dog leg	34.0465	0.0228	262	2185	48.6808	75.4071	5484	45711
	bfgs	35.5586	6.2703	236	8350	10.9842	17.288	3353	9994
	l-bfgs	35.3557	0.0172	300	0	6.3674	0.0067	3648	0
16	stochastic	317.2945	7611.0932	300	0	0.9663	0.0	2515	655
	gauss newton	998.8881	1023910.5706	300	0	5.0392	0.0005	4076	655
	dog leg	34.184	0.1414	300	0	150.6226	0.1738	5420	245
	bfgs	33.7952	1.8389	294	116	15.1638	48.7247	3404	655
	l-bfgs	35.8765	0.4586	300	0	11.7834	0.0094	3648	0

Мы взяли для рассмотрения поменьше методов, так как на данном примере методы работают намного дольше, попробуем сравнить новые методы с обычным стохастическим градиентным спуском (опять же, на `batch=10`). Мы построили 30 точек, удовлетворяющих некоторой полиномиальной регрессии, и выше представлены графики средних (опять же, из запусков на пяти начальных точках) значений полученного argmin функции потерь.
На низких степенях полиномиальной регрессии показатели от линейной почти не отличаются, причины их в целом не изменились. Однако на регрессиях степеней 10 и 15 можно наблюдать, что методы `Gauss-Newton` и `Powell's Dog Leg` начинают сыпаться. Это связано с тем, что гессианы в данном случае размеров 11×11 и 16×16 крайне сложновычислимы, и теряют в точности (в целом, близки к overflow на больших значениях). Здесь же наконец-то постепенно начинает появляться смысл в использовании методов `BFGS` и `L-BFGS`: они теперь выдают наиболее точные показатели за сильно меньшее время. Конечно, `L-BFGS` сильно зависит от параметра `m`.