Задание 3. Алгоритмы безусловной нелинейной оптимизации. Методы первого и второго порядка

Варвара Кошман, С4113, 17.11.2019

Задача: аппроксимация сгенерированных данных функцией, полученной
a) методом градиентного спуска
b) методом сопряженных градиентов
c) методом Ньютона
d) алгоритмом Левенберга-Марквардта

Массив зашумленных данных сгенерирован по правилу: $y_k = \alpha x_k + \beta + \delta_k$, $x_k = \frac{k}{1000}$, где $k$=0,…,1000, $\alpha \in (0,1)$, $\beta \in (0,1)$, $\delta_k \sim N(0,1)$. Функция находится путем минимизации функционала: $D(a,b) = \sum_{k=0}^{1000} (F(x_k, a, b) - y_k))^2$ (с точностью $\varepsilon$=0.001)
при
1. $F(x,a,b) = ax + b$ (линейная аппроксимирующая функция);
2. $F(x,a,b) = a1 + bx$ (рациональная аппроксимирующая функция).

**a) метод градиентного спуска**

1) **градиентный спуск с фиксированным шагом** (0.0001) (медленная сходимость, порядка 150 итераций, при большем шаге метод расходился). Заменен на метод наискорейшего спуска (далее)

**Код:**

```
def gradient_descent(functional):
    prev = np.array([np.random.rand(1)[0], np.random.rand(1)[0]])  # starting point
    iterations = 0
    functional_history = []
    lr = 0.0001
    while True:
        df_da = float(diff(functional, (prev[0], prev[1]), (1, 0)))
        df_db = float(diff(functional, (prev[0], prev[1]), (0, 1)))
        gradient = np.array([df_da, df_db])
        curr = prev - lr * gradient
        iterations += 1
        functional_history.append(functional(curr[0], curr[1]))
        if stop_condition(prev, curr):
            break
        else:
            prev = curr
    # plot_functional_history(iterations, functional_history) # функция для проверки
уменьшения значения функционала с течением итераций
    return curr, iterations
```

2) **метод наискорейшего градиентного спуска** (методом одномерной оптимизации (из лабораторной 2) золотого сечения на каждом шаге ищется оптимальный шаг в направлении антиградиента; число итераций по сравнению с градиентным спуском с фиксированным шагом в среднем в 3 раза меньше)

**Код:**

```
def steepest_gradient_descent(functional):
    prev = np.array([np.random.rand(1)[0], np.random.rand(1)[0]])  # starting point
    iterations = 0
    functional_history = []
    while True:
        df_da = diff(functional, (prev[0], prev[1]), (1, 0))
        df_db = diff(functional, (prev[0], prev[1]), (0, 1))
        gradient = np.array([df_da, df_db])

        left1, right1 = golden_section(fun.partial(update_function, param=prev[0],
dfdparam=df_da), (0.00001, 0.001))
        optimal_step_a = 0.5 * (left1 + right1)
        left2, right2 = golden_section(fun.partial(update_function, param=prev[1],
dfdparam=df_db), (0.00001, 0.001))
        optimal_step_b = 0.5 * (left2 + right2)
        optimal_steps = np.array([optimal_step_a, optimal_step_b])

        curr = prev - optimal_steps * gradient
        iterations += 1
        functional_history.append(functional(curr[0], curr[1]))
        if stop_condition(prev, curr):
            break
        else:
            prev = curr
    # plot_functional_history(iterations, functional_history) # функция для проверки
уменьшения значения функционала с течением итераций
    return curr, iterations
```

**d) алгоритм Левенберга-Марквардта**

(параметр, задающий приращение, изначально берется случайно из равномерного распределения, а затем в процессе итераций увеличивается на определеное значение, пока значение функции невязки с новым значением не будет меньше значения со старым)

**Код:**

```
def levenberg_marquardt_algorithm(functional, regression):
    prev = np.array([np.random.rand(1)[0], np.random.rand(1)[0]])  # starting point
    iterations = 0
    lr = np.random.rand(1)[0]
    factor = 2  # damping factor should be > 1
    k = 1  # "some k" according to wiki
    while True:
        while True:
            df_da = diff(regression, (prev[0], prev[1]), (1, 0))
            df_db = diff(regression, (prev[0], prev[1]), (0, 1))
            gradient_f = np.array(
                [[float(df_da[i]) for i in range(len(df_da))], [float(df_db[i]) for i in
range(len(df_da))]])
            j_jT = gradient_f.dot(gradient_f.transpose())
            H_f = j_jT + lr * np.ones((len(j_jT), len(j_jT)))
            f_s = regression(np.array(prev[0]), np.array(prev[1]))
            y = np.concatenate(dataset[1], axis=0)
            error = gradient_f.dot(y - f_s)
            delta = linalg.inv(H_f).dot(error)
```

```
            curr = prev + delta
            iterations += 1
            if functional(curr) < functional(prev):
                break
            else:
                lr = lr * factor ** k
        if stop_condition(prev, curr):
            break
        else:
            prev = curr
    return curr, iterations
```

для **b) метод сопряженных градиентов** и **c) метод Ньютона** взяты готовые реализации

**Общий код:**

```
linear_reg_f = lambda a, b: np.array([a * dataset[0][i] + b for i in
range(len(dataset[0]))])
rational_reg_f = lambda a, b: np.array([a / (1 + b * dataset[0][i]) for i in
range(len(dataset[0]))])
stop_condition = lambda previous, current: np.linalg.norm(previous - current) < epsilon
functional_linear = lambda point: np.sum(
    np.array([np.square(point[0] * dataset[0][i] + point[1] - dataset[1][i][0]) for i in
range(len(dataset[0]))]))
functional_rational = lambda point: np.sum(
    np.array([np.square(point[0] / (1 + point[1] * dataset[0][i]) - dataset[1][i][0])
for i in range(len(dataset[0]))]))
update_function = lambda param, dfdparam, lr: param - lr * dfdparam

# with 2 parameters for taking partial derivatives
functional_linear2 = lambda a, b: np.sum(
    np.array([np.square(a * dataset[0][i] + b - dataset[1][i][0]) for i in
range(len(dataset[0]))]))
functional_rational2 = lambda a, b: np.sum(
    np.array([np.square(a / (1 + b * dataset[0][i]) - dataset[1][i][0]) for i in
range(len(dataset[0]))]))


def main():
    functionals = [(functional_linear, functional_linear2), (functional_rational,
functional_linear2)]
    regressions = [('linear', linear_reg_f), ('rational', rational_reg_f)]
    jac = lambda point: np.array([float(diff(functionals[i][1], (point[0], point[1]),
(1, 0))),
                                  float(diff(functionals[i][1], (point[0], point[1]),
(0, 1)))])
    for i in range(len(functionals)):
        optimal_gd, gd_iterations = steepest_gradient_descent(functionals[i][1])
        python_cg_res = sp.minimize(functionals[i][0],
                                    np.array((np.random.rand(1)[0],
np.random.rand(1)[0])),
                                    method="CG",
                                    options={'gtol': epsilon})
        optimal_cg = python_cg_res.x
        cg_iterations = python_cg_res.nit
        newton_res = sp.minimize(functionals[i][0],
                                 np.array((np.random.rand(1)[0], np.random.rand(1)[0])),
```

```
                                    jac=jac,
                                    method='Newton-CG',
                                    options={'gtol': epsilon})
        optimal_newton = newton_res.x
        newton_iterations = newton_res.nit
        optimal_lm, lm_iterations = levenberg_marquardt_algorithm(functionals[i][0],
regressions[i][1])
        plot_dataset(regressions[i], optimal_gd, optimal_cg, optimal_newton, optimal_lm)
        print("coefficients initial: ", alpha[0], betta[0])
        print("coefficients with Gradient Descent method (for {} regression) ({}
iterations): "
          .format(regressions[i][0], gd_iterations), optimal_gd[0], optimal_gd[1])
        print("coefficients with Conjugate Gradient Descent method (for {} regression)
({} iterations): "
           .format(regressions[i][0], cg_iterations), optimal_cg[0], optimal_cg[1])
        print("coefficients with Newton method (for {} regression) ({} iterations): "
          .format(regressions[i][0], newton_iterations), optimal_newton[0],
optimal_newton[1])
        print("coefficients with Levenberg-Marquardt algorithm (for {} regression) ({}
iterations): "
          .format(regressions[i][0], lm_iterations), optimal_lm[0], optimal_lm[1])
        print()
```

**Вывод программы:**

Случай линейной регрессии:

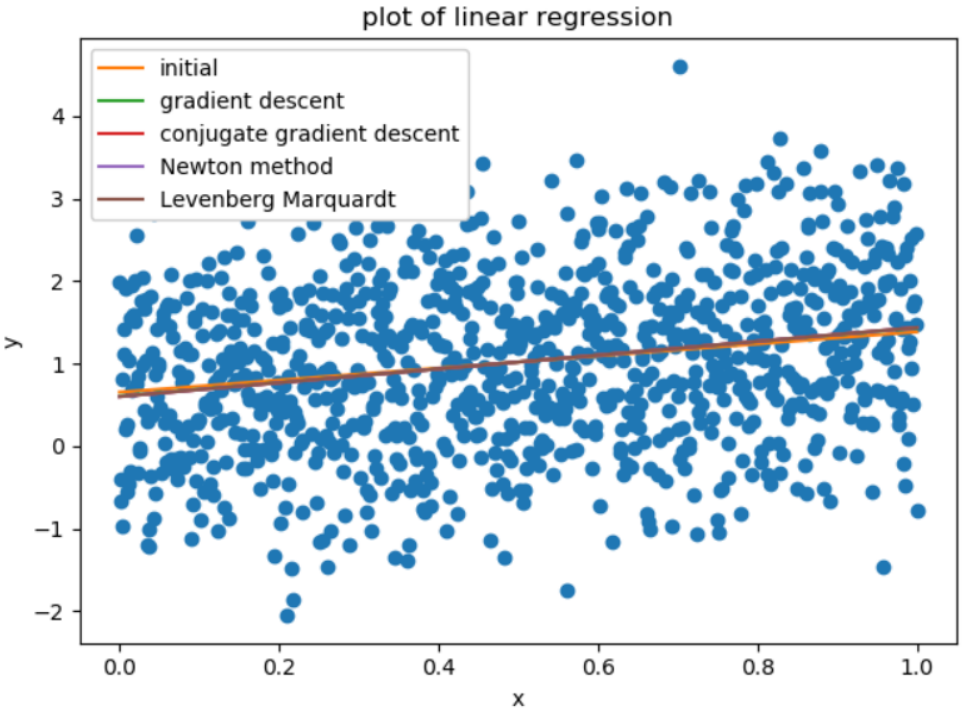
*coefficients initial*: 0.729994031615806 0.6536427502878844

*coefficients with Gradient Descent method* (for linear regression)
(43 iterations): 0.824114609042797 0.607092108377586

*coefficients with Conjugate Gradient Descent method* (for linear regression)
(2 iterations):
0.8361672920129073
0.6006407986708524

*coefficients with Newton method* (for linear regression) (4 iterations):
0.8361673340578842
0.6006407910948717

*coefficients with Levenberg-Marquardt algorithm* (for linear regression) (3 iterations):
0.8361673094904798
0.600640799272505



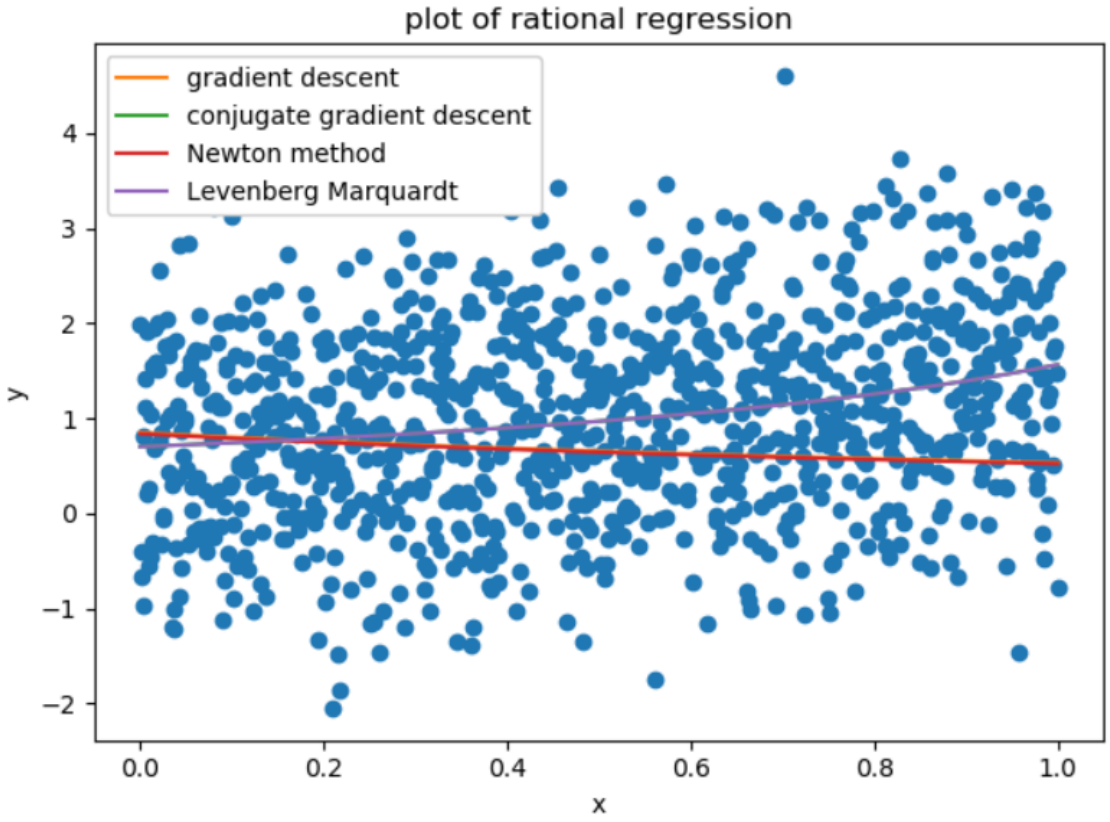
Случай рациональной регрессии:

*coefficients with Gradient Descent method* (for rational regression) (45 iterations):  0.848350068571192 0.594119885190357

*coefficients with Conjugate Gradient Descent method* (for rational regression) (11 iterations): 0.7013579710855257 -0.5513520870742703

*coefficients with Newton method* (for rational regression) (2 iterations):  0.8361673340879896 0.6006407910839753

*coefficients with Levenberg-Marquardt algorithm* (for rational regression) (11 iterations): 0.701376282814439 -0.551332655423163




Выводы: в случае с линейной аппроксимирующей функцией все методы восстановили исходные коэффициенты, с которыми генерировался массив данных, с точностью до шума. Методу сопряженных градиентов достаточного одного шага в направлении сопряженного градиента, чтобы достигнуть минимума, тогда как градиентному методу – 43. Для методов Ньютона и Левенберга-Марквардта достаточно меньше 5 итераций, чтобы сойтись к тому же минимуму. На графике для линейной регрессии видно, что графики всех полученных функций визуально совпадают.

В случае с рациональной регрессией метод Ньютона и метод градиентного спуска сошлись к одним и тем же параметрами, и на графике видно, что графики аппроксимирующих функций для этих методов совпадают. Градиентному методу потребовалось 45 итераций, опять же, этот метод самый долгий. Метод сопряженных градиентов (библиотечная реализация) и алгоритм Левенберга-Марквардта

(ручная реализация) сошлись к одним и тем же минимумам, но отличным от тех, к которым сошлась первая пара методов, при найденных ими параметрах значение функционала оказалось меньше, а значит с этим набором параментров исходные данные должны аппроксимироваться лучше. На графике это видно: графики функций, полученные методами СГ и Л-М совпадают и лучше приближают данные. На сходимость потребовалось в обоих случаях 11 итераций.

**Вспомогательный код:**

```
epsilon = 0.001
# gd_epsilon = 0.0001
alpha = np.random.rand(1)
betta = np.random.rand(1)


def generate_dataset():
    dataset = []
    x = []
    y = []
    for k in range(0, 1001):
        x_k = k / 1000
        y_k = alpha * x_k + betta + np.random.normal(0, 1, 1)[0]
        x.append(x_k)
        y.append(y_k)
    dataset.append(x)
    dataset.append(y)
    return dataset


dataset = generate_dataset()


def golden_section(functional, interval):
    a = interval[0]
    b = interval[1]
    gr = (3 - math.sqrt(5)) / 2
    gr2 = (-3 + math.sqrt(5)) / 2
    x_1 = a + gr * (b - a)
    x_2 = b + gr2 * (b - a)
    y_1 = functional(lr=x_1)
    y_2 = functional(lr=x_2)
    while not (math.fabs(a - b) < epsilon):
        if y_1 < y_2:
            b = x_2
            x_2 = x_1
            y_2 = y_1
            x_1 = a + gr * (b - a)
            y_1 = functional(lr=x_1)
        else:
            a = x_1
            x_1 = x_2
            y_1 = y_2
            x_2 = b + gr2 * (b - a)
            y_2 = functional(lr=x_2)
    return a, b
```

```
def plot_dataset(regression, optimal_gd, optimal_conjugate, optimal_newton, optimal_lm):
    x = dataset[0]
    result_optimal_gd = regression[1](np.array(optimal_gd[0]), np.array(optimal_gd[1]))
    result_optimal_conjugate = regression[1](np.array(optimal_conjugate[0]),
np.array(optimal_conjugate[1]))
    result_optimal_newton = regression[1](np.array(optimal_newton[0]),
np.array(optimal_newton[1]))
    result_optimal_lm = regression[1](np.array(optimal_lm[0]), np.array(optimal_lm[1]))
    plt.plot(x, dataset[1], 'o')
    plt.xlabel('x')
    plt.ylabel('y')
    if regression[0] == 'linear':
        initial = regression[1](alpha, betta)
        plt.plot(x, initial, label='initial')
    plt.plot(x, result_optimal_gd, label='gradient descent')
    plt.plot(x, result_optimal_conjugate, label='conjugate gradient descent')
    plt.plot(x, result_optimal_newton, label='Newton method')
    plt.plot(x, result_optimal_lm, label='Levenberg Marquardt')
    plt.legend(framealpha=1, frameon=True)
    plt.title('plot of {} regression'.format(regression[0]))
    plt.show()


def plot_functional_history(iterations, functional_history):
    fig, ax = plt.subplots(figsize=(12, 8))
    ax.set_ylabel('Value of functional')
    ax.set_xlabel('Iterations')
    _ = ax.plot(range(iterations), np.array(functional_history), 'b.')
    plt.show()
```