Задание 2. Алгоритмы безусловной нелинейной оптимизации. Прямые методы

Варвара Кошман, С4113, 30.10.2019

1. Применение прямых одномерных методов перебора в приложении к регрессионному анализу.

   Поиск x: $f(x) \rightarrow min$ с точностью $\varepsilon = 0.001$

- Метод дихотомии

  Код алгоритма:

```
def dichotomy(function, interval):
a = interval[0]
b = interval[1]
while not (math.fabs(a - b) < epsilon):
        x_1 = (a + b - delta) / 2
        x_2 = (a + b + delta) / 2
        left_value = function(x_1)
        increment_f_calls()
        right_value = function(x_2)
        increment_f_calls()
        if left_value < right_value:
                b = x_1
        else:
                a = x_2
        increment_iterations()
return a, b
```

- Метод золотого сечения

  Код алгоритма:

```
def golden_section(function, 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 = function(x_1)
increment_f_calls()
y_2 = function(x_2)
increment_f_calls()
increment_iterations()
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 = function(x_1)
        else:
```

```
                    a = x_1
                    x_1 = x_2
                    y_1 = y_2
                    x_2 = b + gr2 * (b - a)
                    y_2 = function(x_2)
                increment_f_calls()
                increment_iterations()
        return a, b
```

Полный код программы:

```
import math

epsilon = 0.001
delta = 0.0005

f_1 = lambda x: x ** 3
f_2 = lambda x: math.fabs(x - 0.2)
f_3 = lambda x: x * math.sin(1 / x)

count_f_calls = 0
count_iterations = 0


def increment_f_calls():
    global count_f_calls
    count_f_calls += 1


def zero_f_calls():
    global count_f_calls
    count_f_calls = 0


def increment_iterations():
    global count_iterations
    count_iterations += 1


def zero_iterations():
    global count_iterations
    count_iterations = 0

def main():
    functions = [f_1, f_2, f_3]
    intervals = [[0, 1], [0, 1], [0.1, 1]]
    dichotomy_results = []
    gs_results = []
    for i in range(len(functions)):
        dichotomy_results.append(((dichotomy(functions[i], intervals[i])),count_f_calls,
count_iterations))
        zero_f_calls()
        zero_iterations()
        gs_results.append(((golden_section(functions[i], intervals[i])), count_f_calls,
count_iterations))
        zero_f_calls()
        zero_iterations()
```

```
    print("dichotomy: ", dichotomy_results)
    print("golden section: ", gs_results)


if __name__ == '__main__':
    main()
```

Вывод программы:

dichotomy: [((0, 0.0004770507812500004), 20, 10), ((0.199318359375, 0.19979541015625), 20, 10), ((0.22223583984375003, 0.22261523437500003), 20, 10)]

golden section: [((0, 0.000733137435857404), 17, 16), ((0.199706745025657, 0.2004398824615144), 17, 16), ((0.22226323264212922, 0.2229230563344009), 17, 16)]

Результаты:

	$f(x)=x^3$, $x\in[0,1]$		$f(x)=\|x-0.2\|$, $x\in[0,1]$		$f(x)=x\sin 1x$, $x\in[0.1,1]$	
	# выч-й f	# итераций	# выч-й f	# итераций	# выч-й f	# итераций
Метод дихотомии	20	10	20	10	20	10
Метод золотого сечения	17	16	17	16	17	16

Вывод: Оба метода для всех трех функций получают одинаковые с точностью до 4го разряда точки минимума. Каждый метод для всех трех функций работает одинаковое число итераций: метод дихотомии 10, золотого сечения – 16. Так как метод дихотомии на каждом шаге делит отрезок пополам, то сходится за меньшее число итерациий, чем метод золотого сечения, где отрезок сокращается в пропорции золотого сечения. Но несмотря на это, итоговое число вычисления функции меньше (на каждом шаге, кроме 1го, считается значение f только для 1ой точки).

2. Применение прямых многомерных методов перебора для решения для решения задач линейной и рациональной регрессий

- Метод Гаусса (одномерная оптимизация с помощью прямого перебора)
  Код алгоритма:

```
def minimize_f(regression_f, parameters):
    all_functional_values = {}
    changing = -0.5
    while changing < 1.5:
        functional_value = 0
        for i in range(len(dataset[0])):
            x_k = dataset[0][i]
            y_k = dataset[1][i][0]
            if parameters[0] is None:
                functional_value += np.square(regression_f(x_k, changing,
parameters[1]) - y_k)
            else:
                functional_value += np.square(regression_f(x_k, parameters[0],
changing) - y_k)
```

```
            all_functional_values.update({functional_value: changing})
            changing += epsilon
        optimal_parameter =
    all_functional_values.get(np.min(np.vstack(all_functional_values.keys())))
        return optimal_parameter


    def gauss_method(regression_f):
        a_old = np.random.rand(1)[0]
        b_old = np.random.rand(1)[0]
        iterations = 0
        while True:
            a_new = minimize_f(regression_f, (None, b_old))  # b fixed
            b_new = minimize_f(regression_f, (a_new, None))  # new a fixed
            if stop_condition(a_old, a_new, b_old, b_new):
                break
            else:
                a_old = a_new
                b_old = b_new
                iterations += 1
        return [a_new, b_new], iterations
```

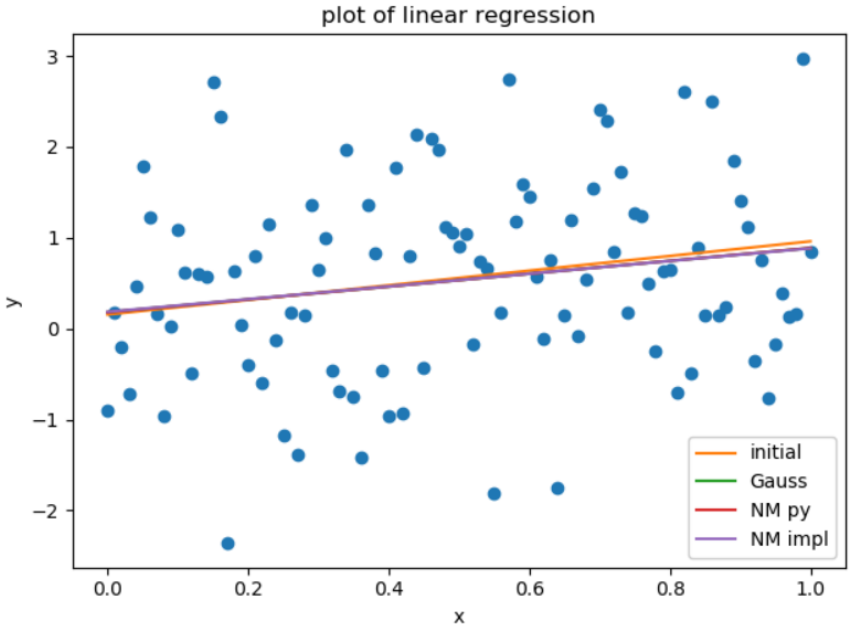
- Метод Нелдера-Мида

  Код алгоритма:

```
    def nelder_mead_method(regression, alpha_p=1, betta_p=0.5, gamma_p=2): \
            # step 1: preparation
        points = [np.array([np.random.rand(1)[0], np.random.rand(1)[0]]) for _ in
    range(3)]
        f_values_vs_points = [np.array([point, functional(point, regression)]) for
    point in points]
        nm = NM(f_values_vs_points)
        iterations = 0
        while True:
            # step 2: sorting
            nm.f_values_vs_points.sort(key=lambda x: x[1])
            f_l = nm.f_values_vs_points[0][1]
            f_g = nm.f_values_vs_points[1][1]
            f_h = nm.f_values_vs_points[2][1]
            x_h = nm.f_values_vs_points[2][0]
            x_g = nm.f_values_vs_points[1][0]
            x_l = nm.f_values_vs_points[0][0]
            # step 3: gravity centre for two points (except max)
            gravity_centre = 0.5 * (x_l + x_g)
            # step 4: reflection
            x_r = (1 + alpha_p) * gravity_centre - alpha_p * x_h
            f_r = functional(x_r, regression)
            # step 5: comparing f_r value with 3 prev points
            if f_r < f_l:
                # good direction, delaying
                x_e = (1 - gamma_p) * gravity_centre + gamma_p * x_r
                f_e = functional(x_e, regression)
                if f_e < f_r:
                    nm.f_values_vs_points[2] = np.array([x_e, f_e])
                else:
                    nm.f_values_vs_points[2] = np.array([x_r, f_r])
```

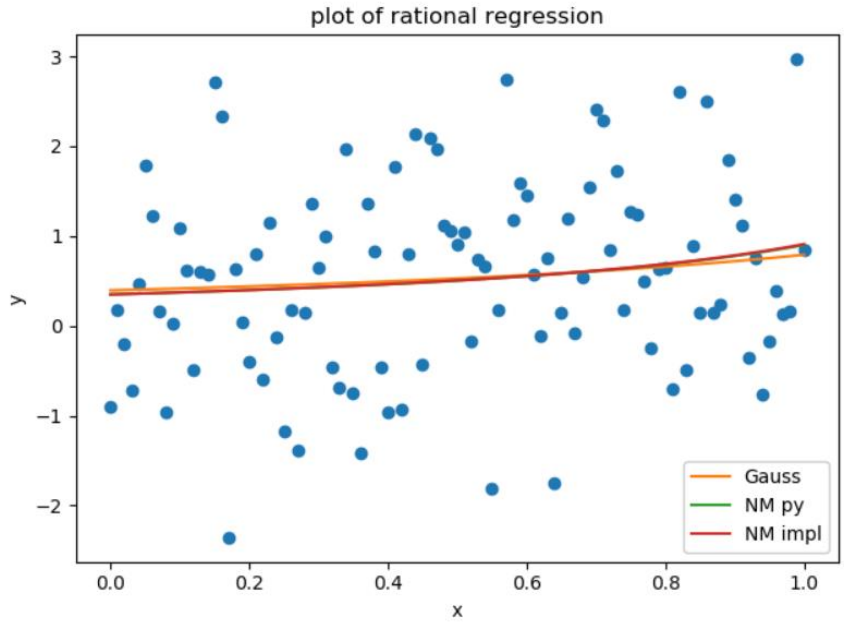
```
        elif f_r < f_g:
            nm.f_values_vs_points[2] = np.array([x_r, f_r])
        elif f_r < f_h:
            nm.f_values_vs_points[2] = np.array([x_r, f_r])
            # step 6: shrinking
            shrinking(regression, nm, betta_p, gravity_centre, x_h,
nm.f_values_vs_points[2][1], x_l, x_g)
        else:
            shrinking(regression, nm, betta_p, gravity_centre, x_h, f_h, x_l,
x_g)
        if is_converged(nm.f_values_vs_points):
            break
        iterations += 1
    return nm.f_values_vs_points[0], iterations


def shrinking(regression, nm, betta_p, gravity_centre, x_h, f_h, x_l, x_g):
    x_s = betta_p * x_h + (1 - betta_p) * gravity_centre
    f_s = functional(x_s, regression)
    if f_s < f_h:
        nm.f_values_vs_points[2] = np.array([x_s, f_s])
    else:
        x_g_new = x_l + 0.5 * (x_g - x_l)
        x_h_new = x_l + 0.5 * (x_h - x_l)
        nm.f_values_vs_points[1] = np.array([x_g_new, functional(x_g_new,
regression)])
        nm.f_values_vs_points[2] = np.array([x_h_new, functional(x_h_new,
regression)])


def is_converged(f_values):
    f_mean = np.sum([f[1] for f in f_values]) / len(f_values)
    sigma = np.sqrt(np.sum([np.square(f_i[1] - f_mean) for f_i in f_values]) /
(len(f_values) - 1))
    if sigma < epsilon:
        return True
    else:
        return False

def functional(arg, regression):
    functional_value = 0
    for i in range(len(dataset[0])):
        x_k = dataset[0][i]
        y_k = dataset[1][i][0]
        functional_value += np.square(regression(x_k, arg[0], arg[1]) - y_k)
    return functional_value
```

Случай линейной регрессии:



Оба метода восстановили параметры линейной регрессионной модели (с точностью до шума, с которым генерировалась выборка). Можно сказать, что линии, полученные методом Гаусса и методом Нелдера-Мида(пакетная и ручная реализации) на графике совпадают.

Случай рациональной регрессии:



Видно, что полученная линия рациональной регрессии лучше приближает исходные данные. Можно сказать, что линии, полученные методом Гаусса и методом Нелдера-Мида(пакетная и ручная реализации) на графике совпадают.

Полный код:

```
import numpy as np
import matplotlib.pyplot as plt
import scipy.optimize as sp
import functools as fun

alpha = np.random.rand(1)
betta = np.random.rand(1)
epsilon = 0.001

linear_reg_f = lambda x, a, b: a * x + b
rational_reg_f = lambda x, a, b: a / (1 + b * x)
stop_condition = lambda a_old, a_new, b_old, b_new: np.linalg.norm(
    np.array([a_old, b_old]) - np.array([a_new, b_new])) < epsilon


class NM(object):
    def __init__(self, f_values_vs_points):
        self.f_values_vs_points = f_values_vs_points


def plot_dataset(regression, optimal_implemented, optimal_python,
optimal_implemented_nm):
    x = dataset[0]
    result_impl_gauss = regression[1](x, np.array(optimal_implemented[0]),
np.array(optimal_implemented[1]))
    result_py_nm = regression[1](x, np.array(optimal_python[0]),
np.array(optimal_python[1]))
    result_impl_nm = regression[1](x, np.array(optimal_implemented_nm[0]),
np.array(optimal_implemented_nm[1]))
    plt.plot(x, dataset[1], 'o')
    plt.xlabel('x')
    plt.ylabel('y')
    if regression[0] == 'linear':
        initial = regression[1](x, alpha, betta)
        plt.plot(x, initial, label='initial')
    plt.plot(x, result_impl_gauss, label='Gauss')
    plt.plot(x, result_py_nm, label='NM py')
    plt.plot(x, result_impl_nm, label='NM impl')
    plt.legend(framealpha=1, frameon=True)
    plt.title('plot of {} regression'.format(regression[0]))
    plt.show()


def generate_dataset():
    dataset = []
    x = []
    y = []
    for k in range(0, 101):
        x_k = k / 100
        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 main():
regressions = [('linear', linear_reg_f), ('rational', rational_reg_f)]
for regression in regressions:
        optimal_implemented_gauss, gauss_iterations = gauss_method(regression[1])
        python_nm_res = sp.minimize(fun.partial(functional, regression=regression[1]),
                            np.array((np.random.rand(1)[0], np.random.rand(1)[0])),
                            method="Nelder-Mead",
                            options={'xtol': 1e-3, 'ftol': 1e-3})
       optimal_implemented_nm, nm_iterations = nelder_mead_method(regression[1])
       plot_dataset(regression, optimal_implemented_gauss, python_nm_res.x,
   optimal_implemented_nm[0])
       print("coefficients initial: ", alpha[0], betta[0])
       print("coefficients with Gauss method (for {} regression) ({} iterations):
   ".format(regression[0],

   gauss_iterations),
             optimal_implemented_gauss[0], optimal_implemented_gauss[1])
       print("coefficients with Nelder-Mead method (for {} regression)({}
   iterations)(python lib): ".format(
           regression[0], python_nm_res.nit),
           python_nm_res.x[0], python_nm_res.x[1])
       print("coefficients with Nelder-Mead method (for {} regression)({} iterations):
   ".format(regression[0],

   nm_iterations),
             optimal_implemented_nm[0], optimal_implemented_nm[1])
       print()


   if __name__ == '__main__':
       main()
```

Вывод программы:

coefficients initial: 0.8054593096392803 0.15523484872804472

coefficients with Gauss method (for linear regression) (21 iterations): 0.706000000000001 0.18100000000000058

coefficients with Nelder-Mead method (for linear regression)(29 iterations)(python lib): 0.7084016414915463 0.17951763658478342

coefficients with Nelder-Mead method (for linear regression)(21 iterations): [0.69918559 0.18497515] 114.61467769354358

coefficients initial: 0.8054593096392803 0.15523484872804472

coefficients with Gauss method (for rational regression) (5 iterations): 0.39600000000000074 -0.5

coefficients with Nelder-Mead method (for rational regression)(38 iterations)(python lib): 0.3504267362251001 -0.6108329684600361

coefficients with Nelder-Mead method (for rational regression)(21 iterations): [0.35000316 -0.61542928] 115.68305416487736

Интересно отметить, что реализованная вручную версия алгоритма Нелдера-Мида достигает необходимую точность за примерно то же число итераций, что метод Гаусса, тогда как библиотечная версия итерируется в среднем в 2 раза дольше (начальные точки брались из равномерного распределения в обоих случаях). Возможно, это связано с разными стратегиями оценки точки останова: библиотечная версия использует в качестве условия останова абсолютную ошибку между точками & между значениями на соседних итерациях, тогда как в ручной имплементации оценивалась дисперсия точек симлекса на итерации.