Jakub Latawiec

Rafał Malik

**Optymalizacja**

Laboratorium - Optymalizacja funkcji wielu zmiennych metodami gradientowymi.

1. **Cel ćwiczenia**

Celem ćwiczenia jest zapoznanie się z gradientowymi metodami optymalizacji poprzez ich implementację oraz wykorzystanie do wyznaczenia minimum podanej funkcji celu.

1. **Przeprowadzenie ćwiczenia**
2. Implementacja metody najszybszego spadku:

solution SD(matrix(\*ff)(matrix, matrix, matrix), matrix(\*gf)(matrix, matrix, matrix), matrix x0, double h0, double epsilon, int Nmax, matrix ud1, matrix ud2)

{

try

{

std::stringstream ss{};

solution XB;

XB.x = x0;

solution XT;

XT = XB;

matrix d;

while (true)

{

if (SAVE\_CHART\_DATA)

ss << XB.x(0) << ";" << XB.x(1) << "\n";

//Wyliczanie kierunku spadku funkcji

XB.grad(gf, ud1, ud2);

d = -XB.g;

//Metoda zmiennokrokowa

if (h0 <= 0)

{

matrix h\_fun\_data(2, 2);

h\_fun\_data.set\_col(XB.x, 0);

h\_fun\_data.set\_col(d, 1);

solution h\_sol = golden(ff, 0, 1, epsilon, Nmax, ud1, h\_fun\_data);

matrix h = h\_sol.x;

XT.x = XB.x + h \* d;

}

//Metoda stałokrokowa

else

{

XT.x = XB.x + h0 \* d;

}

if (solution::g\_calls > Nmax)

{

if (SAVE\_CHART\_DATA)

save\_to\_file("SD\_chart\_h\_" + std::to\_string(h0) + ".csv", ss.str());

XT.fit\_fun(ff, ud1, ud2);

return XT;

}

if (norm(XT.x - XB.x) <= epsilon)

break;

XB = XT;

}

if (SAVE\_CHART\_DATA)

save\_to\_file("SD\_chart\_h\_" + std::to\_string(h0) + ".csv", ss.str());

XT.fit\_fun(ff, ud1, ud2);

return XT;

}

catch (string ex\_info)

{

throw ("solution SD(...):\n" + ex\_info);

}

}

1. Implementacja metody gradientów sprzężonych:

solution CG(matrix(\*ff)(matrix, matrix, matrix), matrix(\*gf)(matrix, matrix, matrix), matrix x0, double h0, double epsilon, int Nmax, matrix ud1, matrix ud2)

{

try

{

std::stringstream ss{};

solution XB;

XB.x = x0;

solution XT;

matrix g\_prev;

matrix g\_curr;

matrix d;

XB.grad(gf, ud1, ud2);

g\_prev = XB.g;

d = -g\_prev;

while (true)

{

ss << XB.x(0) << ";" << XB.x(1) << "\n";

//Metoda zmiennokrokowa

if (h0 <= 0)

{

matrix h\_fun\_data(2, 2);

h\_fun\_data.set\_col(XB.x, 0);

h\_fun\_data.set\_col(d, 1);

solution h\_sol = golden(ff, 0, 1, epsilon, Nmax, ud1, h\_fun\_data);

matrix h = h\_sol.x;

XT.x = XB.x + h \* d;

}

//Metoda stałokrokowa

else

{

XT.x = XB.x + h0 \* d;

}

if (solution::g\_calls > Nmax)

{

if (SAVE\_CHART\_DATA)

save\_to\_file("CG\_chart\_h\_" + std::to\_string(h0) + ".csv", ss.str());

XT.fit\_fun(ff, ud1, ud2);

return XT;

}

if (norm(XT.x - XB.x) <= epsilon)

break;

XT.grad(gf, ud1, ud2);

g\_curr = XT.g;

double beta = pow(norm(g\_curr), 2) / pow(norm(g\_prev), 2);

d = -g\_curr + beta \* d;

g\_prev = g\_curr;

XB = XT;

}

if (SAVE\_CHART\_DATA)

save\_to\_file("CG\_chart\_h\_" + std::to\_string(h0) + ".csv", ss.str());

XT.fit\_fun(ff, ud1, ud2);

return XT;

}

catch (string ex\_info)

{

throw ("solution CG(...):\n" + ex\_info);

}

}

1. Implementacja metody Newtona:

solution Newton(matrix(\*ff)(matrix, matrix, matrix), matrix(\*gf)(matrix, matrix, matrix),

matrix(\*Hf)(matrix, matrix, matrix), matrix x0, double h0, double epsilon, int Nmax, matrix ud1, matrix ud2)

{

try

{

std::stringstream ss{};

solution XB;

XB.x = x0;

solution XT;

matrix d;

while (true)

{

if (SAVE\_CHART\_DATA)

ss << XB.x(0) << ";" << XB.x(1) << "\n";

XB.grad(gf, ud1, ud2);

XB.hess(Hf, ud1, ud2);

d = -inv(XB.H) \* XB.g;

//Metoda zmiennokrokowa

if (h0 <= 0)

{

matrix h\_fun\_data(2, 2);

h\_fun\_data.set\_col(XB.x, 0);

h\_fun\_data.set\_col(d, 1);

solution h\_sol = golden(ff, 0, 1, epsilon, Nmax, ud1, h\_fun\_data);

matrix h = h\_sol.x;

XT.x = XB.x + h \* d;

}

//Metoda stałokrokowa

else

{

XT.x = XB.x + h0 \* d;

}

if (solution::g\_calls > Nmax)

{

if (SAVE\_CHART\_DATA)

save\_to\_file("Newton\_chart\_h\_" + std::to\_string(h0) + ".csv", ss.str());

XT.fit\_fun(ff, ud1, ud2);

return XT;

}

if (norm(XT.x - XB.x) <= epsilon)

break;

XB = XT;

}

if (SAVE\_CHART\_DATA)

save\_to\_file("Newton\_chart\_h\_" + std::to\_string(h0) + ".csv", ss.str());

XT.fit\_fun(ff, ud1, ud2);

return XT;

}

catch (string ex\_info)

{

throw ("solution Newton(...):\n" + ex\_info);

}

}

1. Implementacja metody złotego podziału:

solution golden(matrix(\*ff)(matrix, matrix, matrix), double a, double b, double epsilon, int Nmax, matrix ud1, matrix ud2)

{

try

{

solution Xopt;

double alpha = (pow(5, 0.5) - 1) / 2;

double a0 = a;

double b0 = b;

double c0 = b0 - alpha \* (b0 - a0);

double d0 = a0 + alpha \* (b0 - a0);

do

{

solution c0\_sol;

c0\_sol.x = c0;

c0\_sol.fit\_fun(ff, ud1, ud2);

solution d0\_sol;

d0\_sol.x = d0;

d0\_sol.fit\_fun(ff, ud1, ud2);

if (c0\_sol.y < d0\_sol.y)

{

b0 = d0;

d0 = c0;

c0 = b0 - alpha \* (b0 - a0);

}

else

{

c0 = d0;

a0 = c0;

d0 = a0 + alpha \* (b0 - a0);

}

if (solution::f\_calls > Nmax)

throw std::string("Maximum amount of f\_calls reached!: ");

} while (b0 - a0 > epsilon);

Xopt.x = (a0 + b0) / 2;

return Xopt;

}

catch (string ex\_info)

{

throw ("solution golden(...):\n" + ex\_info);

}

}

1. Implementacja funkcji hipotezy:

double sigmoid(matrix theta, matrix x)

{

return 1.0 / (1.0 + exp(-1.0 \* m2d(trans(theta) \* x)));

}

1. Implementacja testowej funkcji celu:

matrix ff4T(matrix x, matrix ud1, matrix ud2)

{

matrix y;

if (isnan(ud2(0, 0)))

y = pow((x(0) + 2 \* x(1) - 7), 2) + pow((2 \* x(0) + x(1) - 5), 2);

else

y = ff4T(ud2[0] + x \* ud2[1]);

return y;

}

1. Implementacja testowej funkcji gradientu:

matrix gf4T(matrix x, matrix ud1, matrix ud2)

{

matrix y(2, 1);

y(0) = -34.0 + 10.0 \* x(0) + 8.0 \* x(1);

y(1) = -38.0 + 8.0 \* x(0) + 10.0 \* x(1);

return y;

}

1. Implementacja testowej funkcji hessianu:

matrix hf4T(matrix x, matrix ud1, matrix ud2)

{

matrix y(2, 2);

y(0, 0) = 10;

y(0, 1) = 8;

y(1, 0) = 8;

y(1, 1) = 10;

return y;

}

1. Implementacja funkcji celu dla problemu rzeczywistego:

matrix ff4R(matrix theta, matrix X, matrix Y)

{

matrix y;

double sum = 0.0;

for (int i = 0; i < 100; ++i)

{

double y\_i = Y[i](0);

matrix x\_i = X[i];

sum += y\_i \* log(sigmoid(theta, x\_i)) + (1.0 - y\_i) \* log(1.0 - sigmoid(theta, x\_i));

}

y = (-1.0 / 100.0) \* sum;

return y;

}

1. Implementacja funkcji gradientu dla problemu rzeczywistego:

matrix gf4R(matrix theta, matrix X, matrix Y)

{

matrix y(3, 1);

for (int j = 0; j < 3; ++j)

{

double sum = 0.0;

for (int i = 0; i < 100; ++i)

{

double y\_i = Y[i](0);

matrix x\_i = X[i];

sum += (sigmoid(theta, x\_i) - y\_i) \* x\_i(j);

}

y(j) = (1.0 / 100.0) \* sum;

}

return y;

}

1. Implementacja funkcji lab4():

void lab4()

{

#ifdef SAVE\_TO\_FILE

create\_environment("lab04");

#endif

//Dane dokładnościowe

double epsilon = 1E-4;

int Nmax = 10000;

#ifdef CALC\_TEST

//Generator liczb losowych

std::random\_device rd;

std::mt19937 gen(rd());

std::uniform\_real\_distribution<> x0\_dist(-10.0, 10.0);

//Stringstream do zapisu danych

std::stringstream test\_ss;

//Rozwiązanie dla wyników testowych

solution test\_sol;

//Punkty startowe

matrix test\_x0{};

//Długości kroków

double h0\_arr[] = { 0.05, 0.12, 0.0 };

for (auto& h0 : h0\_arr)

{

for (int i = 0; i < 100; ++i)

{

test\_x0 = matrix(2, new double[2] {x0\_dist(gen), x0\_dist(gen)});

test\_ss << test\_x0(0) << ";" << test\_x0(1) << ";";

test\_sol = SD(ff4T, gf4T, test\_x0, h0, epsilon, Nmax);

test\_ss << test\_sol.x(0) << ";" << test\_sol.x(1) << ";" << m2d(test\_sol.y) << ";" << test\_sol.f\_calls << ";" << test\_sol.g\_calls << ";";

solution::clear\_calls();

test\_sol = CG(ff4T, gf4T, test\_x0, h0, epsilon, Nmax);

test\_ss << test\_sol.x(0) << ";" << test\_sol.x(1) << ";" << m2d(test\_sol.y) << ";" << test\_sol.f\_calls << ";" << test\_sol.g\_calls << ";";

solution::clear\_calls();

test\_sol = Newton(ff4T, gf4T, hf4T, test\_x0, h0, epsilon, Nmax);

test\_ss << test\_sol.x(0) << ";" << test\_sol.x(1) << ";" << m2d(test\_sol.y) << ";" << test\_sol.f\_calls << ";" << test\_sol.g\_calls << ";" << test\_sol.H\_calls << "\n";

solution::clear\_calls();

}

#ifdef SAVE\_TO\_FILE

save\_to\_file("test\_h\_" + std::to\_string(h0) + ".csv", test\_ss.str());

#endif

//Czyszczenie zawartoci ss

test\_ss.str(std::string());

}

//Liczenie danych do wykresów

SAVE\_CHART\_DATA = true;

test\_x0 = matrix(2, new double[2] {-5.98, -2.31});

for (auto& h0 : h0\_arr)

{

test\_sol = SD(ff4T, gf4T, test\_x0, h0, epsilon, Nmax);

solution::clear\_calls();

test\_sol = CG(ff4T, gf4T, test\_x0, h0, epsilon, Nmax);

solution::clear\_calls();

test\_sol = Newton(ff4T, gf4T, hf4T, test\_x0, h0, epsilon, Nmax);

}

SAVE\_CHART\_DATA = false;

#endif

#ifdef CALC\_SIMULATION

//Pobieranie danych z pliku

matrix x\_data = file\_reader::fileToMatrix(3, 100, "../input\_data/lab04/XData.txt");

matrix y\_data = file\_reader::fileToMatrix(1, 100, "../input\_data/lab04/YData.txt");

//Początkowe theta

matrix theta\_start = matrix(3, new double[3] {0.0, 0.0, 0.0});

//Długości kroków

double h0\_arr\_sim[] = { 0.01, 0.001, 0.0001 };

for (auto& h0 : h0\_arr\_sim)

{

//Wyliczanie optymalnych parametrów klasyfikatora

solution theta\_opt = CG(ff4R, gf4R, theta\_start, h0, epsilon, Nmax, x\_data, y\_data);

std::cout << theta\_opt << "\n";

//Obliczanie procentu dostających się osób

double percentage = 0;

for (int i = 0; i < 100; ++i)

{

matrix curr\_x = x\_data[i];

matrix curr\_y = y\_data[i];

double calculated\_value = sigmoid(theta\_opt.x, curr\_x);

if (round(calculated\_value) == curr\_y)

++percentage;

}

std::cout << "Percentage: " << percentage << "\n\n\n";

solution::clear\_calls();

}

#endif

}

1. **Parametry algorytmów**

Dokładność (epsilon): 1E-4

Maksymalna ilość wywołań funkcji (Nmax): 10 000

1. **Dyskusja wyników**
2. Wyniki testowe:

Dla długości kroku równej 0.05 każda z metod znalazła minimum w okolicach: x1 = 1, x2 = 3. Najmniejszą liczbę wywołań funkcji gradientu otrzymano metodą gradientów sprzężonych (jest to ok 29 wywołań). Dla Kroku równego 0.12 metoda najszybszego spadku nie znalazła żadnego rozwiązania. Wynika to z „przeskakiwania” znajdowanych punktów między naszym minimum a późniejszego dążenia do +- nieskończoności. Takim samym zachowaniem cechuje się metoda gradientów sprzężonych ale tylko dla niektórych przypadków (podczas wyliczania średnich w tabeli 2 w pliku .xlsx, takie wyniki były pomijane). Metoda Newtona wyliczyła minimum niezależnie od punktu startowego. Wszystkie metody wyliczane ze zmiennym krokiem znalazły odpowiednie minimum. Zmniejszyliśmy tym ilość wywołań funkcji gradientu ale kosztem wzrostu ilości wywołań f\_calls (spowodowane wywoływaniem funkcji golden() wyliczającej optymalny krok). Przedstawienie działania metod iteracja po iteracji znajduje się w pliku .xlsx w zakładce „Wykresy”. Wykresy pokazują jak trzy powyższe metody zbiegają się. (w przypadku metody najszybszego spadku dla kroku 0.12, pokazano tylko 5 pierwszych iteracji dla czytelności wykresu).

1. Wyniki problemu rzeczywistego:

Dla kroku 0.01 parametry klasyfikatora są skutkiem zakończenia programu przez przekroczenie ilości g\_calls (minimum zostało „przeskoczone” i dążyliśmy do +- nieskończoności). Dla kroku 0.001 znalezione parametry klasyfikatora wynoszą [-11.74, 0.12, 0.13] a wartość funkcji kosztu wynosi 0.27 (im mniejsza tym lepiej dopasowany model). Dla kroku 0.0001 minimum zostało znalezione za wcześnie, przez co znalezione parametry klasyfikatora i wartość funkcji kosztu są gorsze niż w przypadku kroku równego 0.001 (wartość funkcji kosztu wynosi 0.53). Wyznaczona granica klasyfikacji wynosi zatem:

1. **Wnioski**

Za pomocą metod gradientowych i podanych danych stworzyliśmy model, dzięki któremu jesteśmy w stanie wyznaczyć granicę klasyfikacji osób na studia. Model nie jest jednak na tyle dokładny aby wdrożyć go na uczelnie. Aby to zrobić należałoby znaleźć optymalny krok, który znalazłby lepsze parametry klasyfikatora.