Optymalizacja IT gr. 2

Dariusz Homa

Wojciech Jurgielewicz

Michał Koleżyński

Projekt 4

Cel ćwiczenia

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

Wykonanie

Kody funkcji

Metoda 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
    {
        solution Xopt;
        Xopt.x = x0;
        solution X_prev;
        matrix d;
        double hi;
        do
        {
            X_prev = Xopt;
            Xopt.grad(gf, ud1, ud2);
            d = -Xopt.g;

            if (h0 <= 0)
            {
                matrix h_fun_data(2, 2);
                h_fun_data.set_col(X_prev.x, 0);
                h_fun_data.set_col(d, 1);
                solution h_sol = golden(ff, 0, 1, epsilon, Nmax, ud1,
h_fun_data);
                solution::f_calls = 0;
                matrix h = h_sol.x;
                Xopt.x = X_prev.x + h * d;
            }
            else
            {
                Xopt.x = X_prev.x + h0 * d;
            }
            if (solution::g_calls > Nmax)
                throw std::string("Przekroczono limit wywolan funkcji grad()
:(");

        } while (norm(Xopt.x - X_prev.x) > epsilon);

        Xopt.fit_fun(ff, ud1, ud2);
        return Xopt;
    }
    catch (string ex_info)
    {
        throw("solution SD(...):\n" + ex_info);
    }
}
```

Metoda 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
    {
        solution Xopt = x0;
        solution X_prev;

        Xopt.grad(gf, ud1, ud2);

        matrix d = -Xopt.g;

        matrix di;

        do
        {
            X_prev = Xopt;
            d = di;

            Xopt.grad(gf, ud1, ud2);
            double beta = pow(norm(Xopt.g), 2) / pow(norm(X_prev.g), 2);

            di = -Xopt.g + beta * d;

            if (h0 <= 0)
            {
                //zmiennokrokowa
                matrix h_fun_data(2, 2);
                h_fun_data.set_col(X_prev.x, 0);
                h_fun_data.set_col(di, 1);
                solution h_sol = golden(ff, 0, 1, epsilon, Nmax, ud1,
h_fun_data);
                solution::f_calls = 0;
                matrix h = h_sol.x;
                Xopt.x = X_prev.x + h * di;
            }
            else
            {
                //stalokrokowa
                Xopt.x = X_prev.x + h0 * di;
            }
            if (solution::g_calls > Nmax)
                throw std::string("Przekroczono limit wywolan funkcji grad()
:(");

        } while (norm(Xopt.x - X_prev.x) > epsilon);

        Xopt.fit_fun(ff, ud1, ud2);
```

```
return Xopt;
    }
    catch (string ex_info)
    {
        throw("solution CG(...):\n" + ex_info);
    }
```

Metoda 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
    {
        solution Xopt = x0;
        solution X_prev;

        matrix d;

        do
        {
            X_prev = Xopt;

            Xopt.hess(Hf, ud1, ud2);
            Xopt.grad(gf, ud1, ud2);

            d = -inv(Xopt.H) * Xopt.g;

            if (h0 <= 0)
            {
                matrix h_fun_data(2, 2);
                h_fun_data.set_col(X_prev.x, 0);
                h_fun_data.set_col(d, 1);
                solution h_sol = golden(ff, 0, 1, epsilon, Nmax, ud1,
h_fun_data);
                solution::f_calls = 0;
                matrix h = h_sol.x;
                Xopt.x = X_prev.x + h * d;
            }
            else
            {
                Xopt.x = X_prev.x + h0 * d;
            }
            if (solution::H_calls > Nmax)
                throw std::string("Przekroczono limit wywolan funkcji hess()
:(");

            if (solution::g_calls > Nmax)
                throw std::string("Przekroczono limit wywolan funkcji grad()
:(");

        } while (norm(Xopt.x - X_prev.x) > epsilon);

        Xopt.fit_fun(ff, ud1, ud2);
        return Xopt;
    }
```

Metoda 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 = (sqrt(5.0) - 1.0) / 2.0;

        double c0 = b - alpha * (b - a);
        double d0 = a + alpha * (b - a);

        do {
            solution c;
            c.x = c0;
            c.fit_fun(ff, ud1, ud2);

            solution d;
            d.x = d0;
            d.fit_fun(ff, ud1, ud2);

            if (c.y < d.y)
            {
                b = d0;
                d0 = c0;
                c0 = b - alpha * (b - a);
            }
            else
            {
                c0 = d0;
                a = c0;
                d0 = a + alpha * (b - a);
            }

            if (solution::f_calls > Nmax)
                throw std::string("Przekroczono limit wywolan funkcji golden
:(");

        } while (b - a > epsilon);

        Xopt = (a + b) / 2;
        return Xopt;
    }
    catch (string ex_info)
    {
        throw("solution golden(...):\n" + ex_info);
    }
}
```

Testowa funkcja celu

$$f(x_1, x_2) = (x_1 + 2x_2 - 7)^2 + (2x_1 + x_2 - 5)^2$$

Ograniczenia:

$$x_1^0 \in [-10,10], x_2^0 \in [-10,10]$$

Cel

- wykonanie 100 optymalizacji dla każdej długości kroku startując z losowego punktu x_1^0 oraz x_2^0. Znalezienie minimum funkcji celu.

Parametry

- dokładność $\epsilon = 10^{-8}$
- maksymalna liczba wywołań funkcji $N_{max} = 10^4$
- długości kroku: dla wersji stałokrokowej $h \in \{0.05, 0.12\}$ oraz dla wersji zmiennokrokowej $h \in \{0\}$

Funkcja celu

```
matrix lab4_fun(matrix x, matrix ud1, matrix ud2)
{
    return pow((x(0) + 2 * x(1) - 7), 2) + pow((2 * x(0) + x(1) - 5), 2);
}

// funkcja testowa
matrix fT4(matrix x, matrix ud1, matrix ud2)
{
    matrix y;

    if (isnan(ud2(0, 0)))
    {
        y = lab4_fun(x, ud1, ud2);
    }
    else
    {
        y = fT4(ud2[0] + x * ud2[1]);
    }


    return y;
}

matrix lab4_grad(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;
}

matrix lab4_hes(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;
}
```

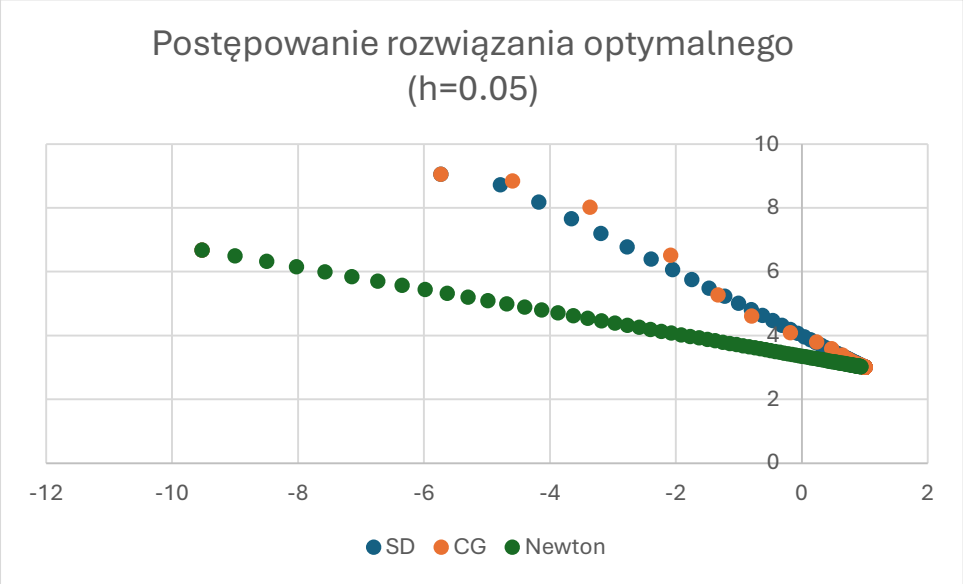
Kod

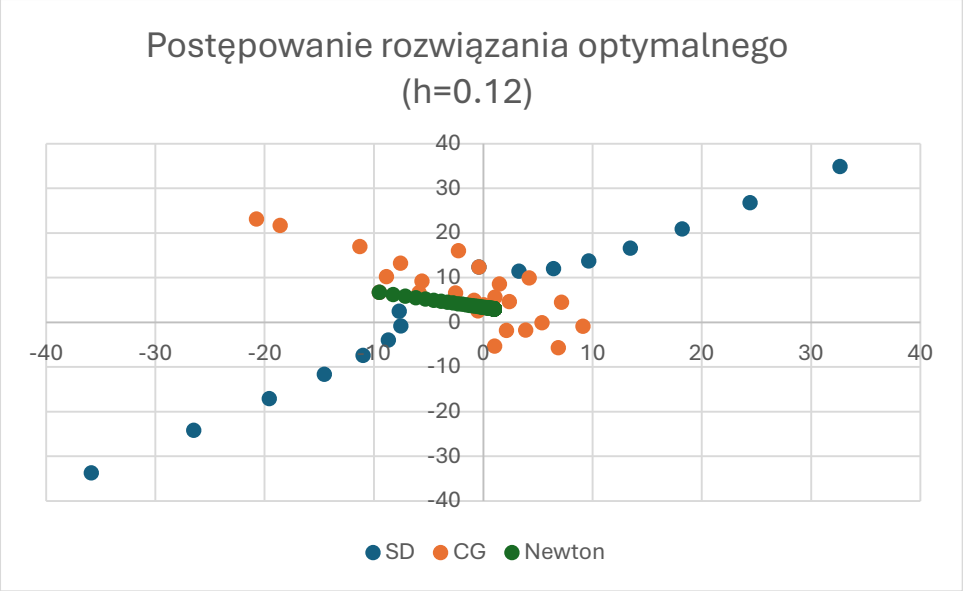
100 optymalizacji w losowych punktach startowych x_1^0, x_2^0

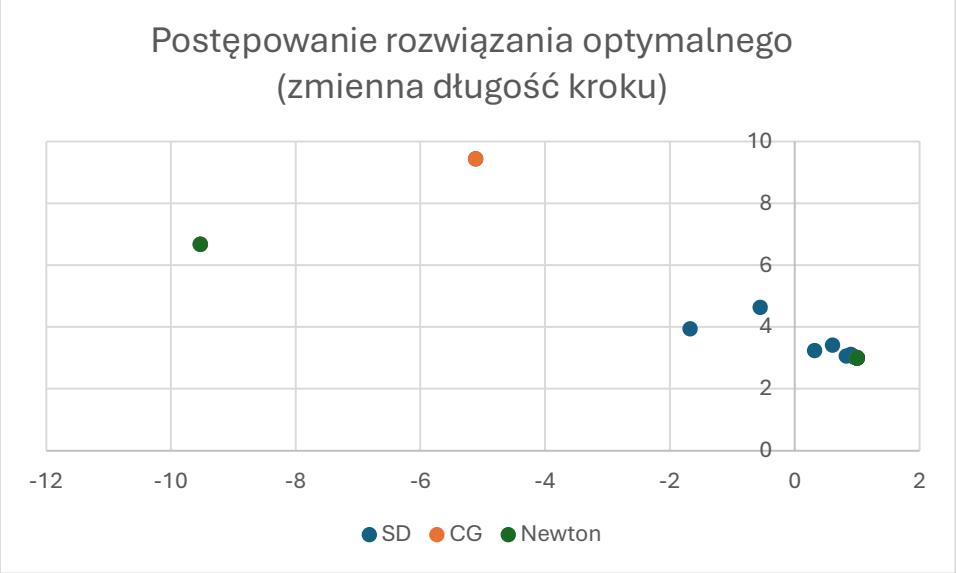
```
for (auto h : hi)
    {
        std::cout << "\nh: " << h << "\n\n";
        for (int i = 0; i < 100; i++) {
            x0(0) = RandomNumberGenerator::Get().Double(-10, 10);
            x0(1) = RandomNumberGenerator::Get().Double(-10, 10);
            SAVE_TO_FILE("x-" + std::to_string(h) + ".txt") << x0(0) << ";" <<
x0(1) << "\n";
            solution simplegrad = SD(fT4, lab4_grad, x0, h, epsilon, nmax);
            SAVE_TO_FILE("SD-" + std::to_string(h) + ".txt") <<
simplegrad.x(0) << ";" << simplegrad.x(1) << ";" << simplegrad.y(0) << ";" <<
solution::f_calls << ";" << solution::g_calls << "\n";
            solution::clear_calls();

            solution complexgrad = CG(fT4, lab4_grad, x0, h, epsilon, nmax);
            SAVE_TO_FILE("CG-" + std::to_string(h) + ".txt") <<
complexgrad.x(0) << ";" << complexgrad.x(1) << ";" << complexgrad.y(0) << ";"
<< solution::f_calls << ";" << solution::g_calls << "\n";
            solution::clear_calls();


            solution newton = Newton(fT4, lab4_grad, lab4_hes, x0, h, epsilon,
nmax);
            SAVE_TO_FILE("Newton-" + std::to_string(h) + ".txt") <<
newton.x(0) << ";" << newton.x(1) << ";" << newton.y(0) << ";" <<
solution::f_calls << ";" << solution::g_calls << ";" << solution::H_calls  <<
"\n";
            solution::clear_calls();
        }

    }
```

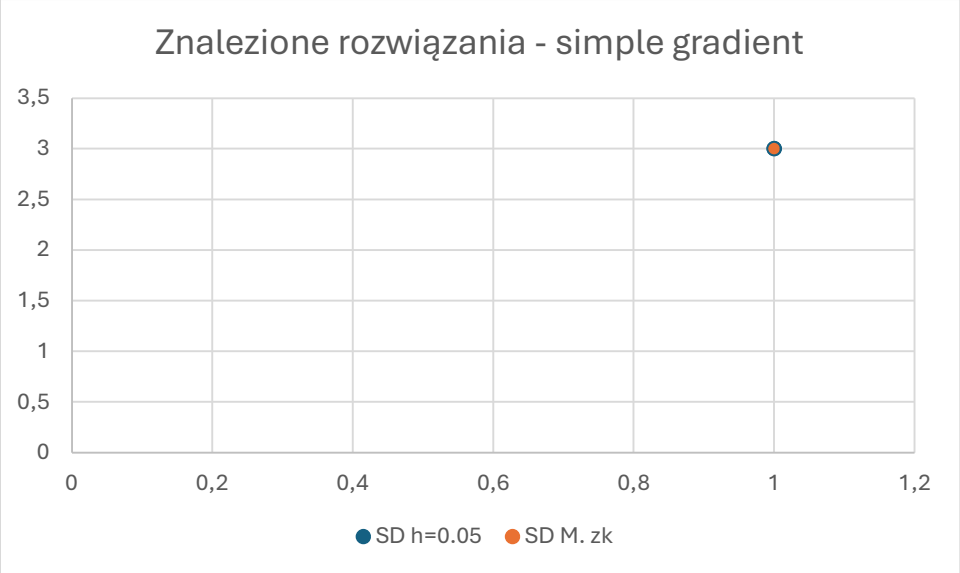
Analiza szybkości znalezienia minimum, dla każdej metody i każdego kroku dla punktu startowego $x_1^0 = -9.5288$ oraz $x_2^0 = 6.6786$

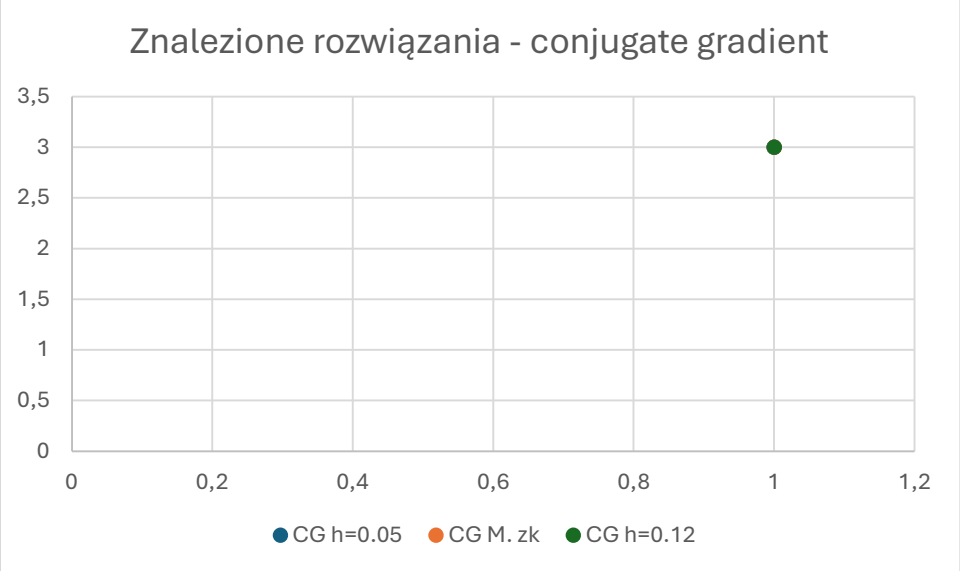
```
        x0(0) = -9.5288;
        x0(1) = 6.6786;

        for (auto h : hi) {
            std::cout << "Simple gradient:\n";
            solution simplegrad = SD(fT4, lab4_grad, x0, h, epsilon, nmax);
            solution::clear_calls();

            std::cout << "Complex gradient:\n";
            solution complexgrad = CG(fT4, lab4_grad, x0, h, epsilon, nmax);
            solution::clear_calls();

            std::cout << "Newton:" << h << "\n";
            solution newton = Newton(fT4, lab4_grad, lab4_hes, x0, h, epsilon,
nmax);
            solution::clear_calls();
        }
```

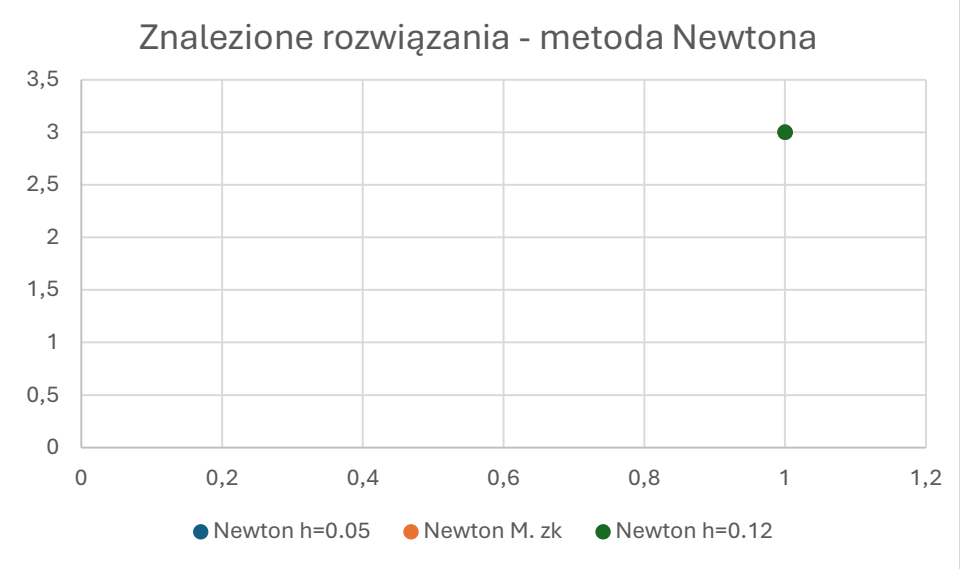



Postępowanie rozwiązania optymalnego
(h=0.12)
40
30
20
10
0
-10
-20
-30
-40
-40
-30
-20
-10
0
10
20
30
40
SD
CG
Newton


Postępowanie rozwiązania optymalnego
(zmienna długość kroku)
10
8
6
4
2
0
-12
-10
-8
-6
-4
-2
0
2
SD
CG
Newton


Znalezione rozwiązania - simple gradient
3,5
3
2,5
2
1,5
1
0,5
0
0
0,2
0,4
0,6
0,8
1
1,2
SD h=0.05
SD M. zk


Znalezione rozwiązania - conjugate gradient
3,5
3
2,5
2
1,5
1
0,5
0
0
0,2
0,4
0,6
0,8
1
1,2
CG h=0.05
CG M. zk
CG h=0.12


Znalezione rozwiązania - metoda Newtona
3,5
3
2,5
2
1,5
1
0,5
0
0
0,2
0,4
0,6
0,8
1
1,2
Newton h=0.05
Newton M. zk
Newton h=0.12

Omówienie wyników

- Dla długości kroku $h = 0.12$ metoda szybkiego spadku nie potrafi znaleźć minimum funkcji. Znacznie lepiej radzi sobie z tym metoda gradientów sprzężonych gdzie tylko dla niektórych punktów startowych algorytm nie potrafi znaleźć wyniku. Żadnego problemu z tym nie ma natomiast metoda Newtona.
- Metoda zmiennokrokowa, gdzie h = 0, pokazuje znaczny wzrost wydajności względem metody stałokrokowej. Dla metody najszybszego spadku i gradientów sprzężonych jest to wzrost kilku-, kilkunastokrotny , a dla metody Newtona nawet kilkusetkrotny względem optymalizacji dla kroku $h = 0.05$
- Każda metoda optymalizacji dla tej samej długości kroku znajduje różną wartość w minimum funkcji. Ze względu na to że w każdym przypadku jest to liczba co najmniej rzędu $10e^{-12}$ możemy założyć, że jest to w przybliżeniu zero.

Problem rzeczywisty

Cel

- Wykonanie jednej optymalizacji metodą gradientów sprzężonych dla każdej długości kroku startując z punktu $\theta^0 = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$.
- Znalezienie optymalnych parametrów klasyfikatora, który na podstawie otrzymanych ocen, będzie w stanie przewidzieć, czy dana osoba zostanie przyjęta na uczelnie.

Parametry

- dokładność $\epsilon = 10^{-4}$
- maksymalna liczba wywołań funkcji $N_{max} = 10^4$
- długości kroku $h \in \{0.01, 0.001, 0.0001\}$

Kod

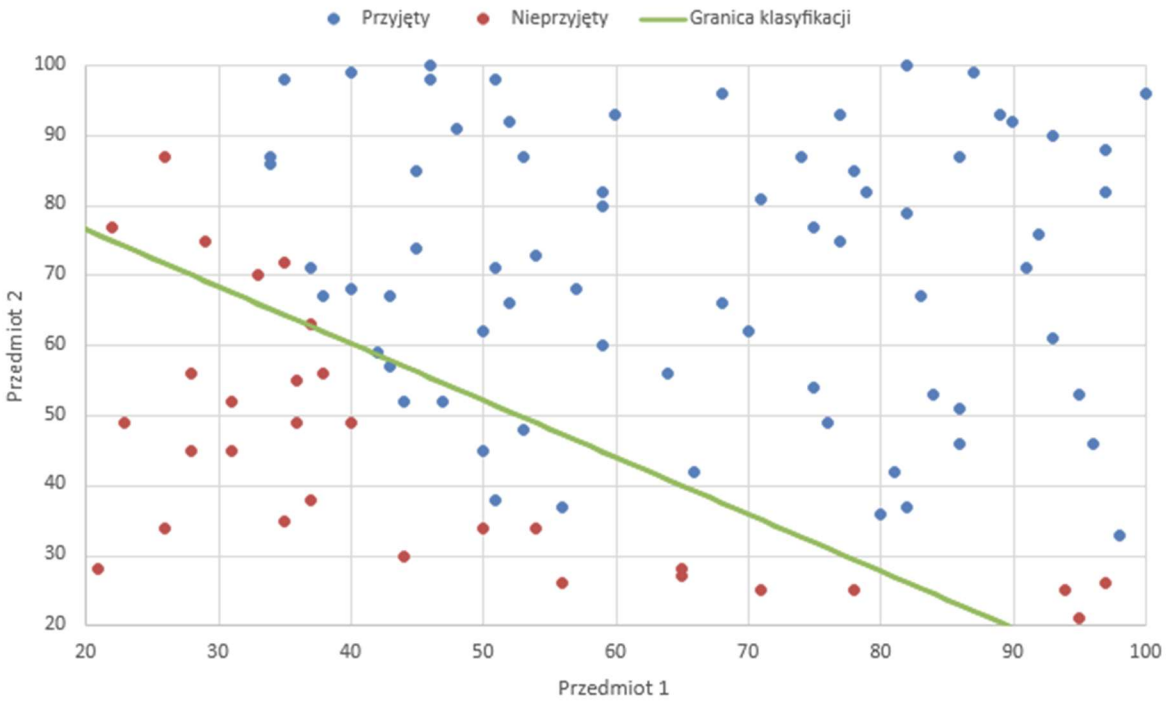
```
double sigmoid(matrix theta, matrix x)
{
    matrix coef = trans(theta) * x;
    return 1.0 / (1.0 + exp(-m2d(coef)));
}

matrix cost_function(matrix theta, matrix ud1, matrix ud2)
{
    constexpr int m = 100;

    double sum = 0.0;
    for (int i = 0; i < m; i++)
        sum += ud2(i) * log(sigmoid(theta, trans(get_row(ud1, i)))) + (1 -
ud2(i)) * log(1 - sigmoid(theta, trans(get_row(ud1, i))));

    sum = sum * -1.0 / m;
    return sum;
}

matrix cost_function_grad(matrix theta, matrix ud1, matrix ud2)
{
    constexpr int m = 100;

    constexpr int n = 3;

    matrix result(n, 1);

    for (int j = 0; j < n; j++)
    {
        double sum = 0.0;
        for (int i = 0; i < m; i++)
            sum += (sigmoid(theta, trans(get_row(ud1, i))) - ud2(i)) * ud1(i,
j);

        result(j) = sum * 1.0 / m;
    }
    return result;
```

Wywołanie:

```
        double hi[] = { 1e-4, 1e-3, 1e-2 };
        double epsilon = 1e-4;
        matrix theta(3, new double[] { 0, 0, 0 });

        for (auto h : hi)
        {
            std::cout << "\nh: " << h << "\n\n";

            solution result = CG(cost_function, cost_function_grad, theta, h,
epsilon, nmax, data->x, data->y);
            std::cout << result << "\n";
            solution::clear_calls();

            int p = 0;
            for (int i = 0; i < 100; i++)
            {
                matrix x = get_row(data->x, i);
                double pi = round(sigmoid(result.x, trans(x)));
                p += (pi == data->y(i) ? 1 : 0);
            }
            cout << "P(0*) = " << p << "\n";
        }
```

Omówienie wyników

Dla długości kroku 0,01 metoda gradientów sprzężonych się nie zbiega w tym przypadku, dlatego dane tylko otrzymano dla kroków 0,001 i 0,0001. Z tabeli "klasyfikator" można wywnioskować, że dla większego kroku model nauczony jest lepiej, gdyż $P(\theta *)$jest wtedy największe, jednak zaś dla zbyt wielkiego kroku wartość się nie zbiega. Optymalna długość kroku w tym przypadku jest pomiędzy 0,01 i 0,001.



Granica klasyfikacji dla obliczonych optymalnych wartości θ jest w stanie z dużą precyzją przewidzieć, czy student zostanie przyjęty na uczelnie - dzieli punkty na przyjętych (nad granicą) i nieprzyjętych (pod granicą). Model jest ograniczony przez fakt, że granica klasyfikacji jest funkcją liniową, jednak z wykresu widać, że liczba punktów przyjętych pod granicą jest podobna do liczby punktów nieprzyjętych nad granicą. Można więc stwierdzić, że jak na model liniowy jest bardzo trafny.

Wnioski

- Metody gradientowe optymalizacji funkcji wielu zmiennych są bardzo czułe na długość kroku, gdyż dla niepoprawnie dobranego mogą nie zbiegać się do wyniku. Zależy to również od kształtu konkretnej funkcji, którą optymalizujemy, zatem dobór rozmiaru kroku jest nietrywialny.
- Metoda najszybszego spadku jest powolna względem pozostałych dwóch metod, oraz w zależności od rozmiaru kroku nie zbiega się.
- Metoda gradientów sprzężonych działa najszybciej spośród trzech wybranych metod dla stałego kroku, jednak jak widać w problemie rzeczywistym również się nie zbiega dla pewnych długości kroku.

- Jak widać z wykresu dla kroku 0,12, metody gradientów sprzężonych oraz najszybszego spadku “przeskakują” optimum dla większej wartości kroku, co jest powodem okazjonalnych niezbieżności wyników. Natomiast metoda Newtona stabilnie kroczy w stronę optimum, przez co w teorii powinna być w stanie znaleźć optimum dla dowolnego rozmiaru kroku. Metoda Newtona jest powolniejsza od pozostałych dla kroku stałego, jednak jest najszybsza dla kroku zmiennego.
- Z powyższych spostrzeżeń wynika, że metoda Newtona ze zmiennym krokiem jest najlepszą z pośród trzech implementowanych metod gradientowych.