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.

2. **Przeprowadzenie ćwiczenia**
   a) 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);
    }
}
```

b) 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);
        }
}
```

c) 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);
    }
}
```

d) 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);
	}
}
```

e) Implementacja funkcji hipotezy:

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

f) 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;
}
```

g) 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;
}
```

h) 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;
}
```

i) 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;
}
```

j) 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;
}
```

k) 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
}
```

3. Parametry algorytmów

Dokładność (epsilon): 1E-4
Maksymalna ilość wywołań funkcji (Nmax): 10 000

4. Dyskusja wyników

a) 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).

b) 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:

$$x_2 = -\frac{-11.7373 + 0.102798 * x_1}{0.126503}$$

5. 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.