Национальный исследовательский университет
информационных технологий, механики и оптики

Кафедра Компьютерных технологий и программирования

Лабораторная работа №3

Выполнили:
Дроботов И.В. М 3232
Бандурин В.А. М 3232
Ларский Н.А. М 3232

Преподаватель:
Свинцов М.В.

Санкт-Петербург
2023

Содержание

1 Цель

Цель данного исследования заключается в реализации двух методов, а именно Gauss-Newton и Powell Dog Leg, для решения задачи нелинейной регрессии. Далее необходимо сравнить эффективность этих методов с уже реализованными в предыдущих работах. Кроме того, в рамках исследования предусмотрена реализация метода BFGS и его анализ на сходимость при минимизации различных функций.

Постановка задачи

1. Реализовать методы Gauss-Newton и Powell Dog Leg для решения нелинейной регрессии. Сравнить эффективность с методами реализованными в предыдущих работах.

2. Реализовать метод BFGS и исследовать его сходимость при минимизации различных функций. Сравнить с другими реализованными методами.

3. Реализовать и исследовать метод L-BFGS.

2 Решение

2.1 Метод Гаусса-Ньютона

Метод Гаусса-Ньютона - это способ решения задачи наименьших квадратов, таких как полиномиальная регрессия. Вместо использования полиномиальной функции напрямую, мы приближаем ее линейной комбинацией базисных функций. Базисные функции выбираются таким образом, чтобы они представляли разные степени полинома и могли быть легко комбинированы. Мы хотим найти коэффициенты модели, которые минимизируют сумму квадратов разницы между наблюдаемыми значениями и значениями, предсказанными моделью.

Метод Гаусса-Ньютона основан на итерациях. На каждой итерации мы линеаризуем модель и используем метод Гаусса-Ньютона для обновления коэффициентов модели. Мы выражаем ошибку модели как линейную комбинацию производных базисных функций. Затем мы формулируем задачу в матричной форме и решаем систему уравнений, чтобы найти новые значения коэффициентов модели. Этот процесс повторяется до достижения сходимости и получения оптимальных коэффициентов модели.

Рис. 1. Код Гаусса-Ньютона

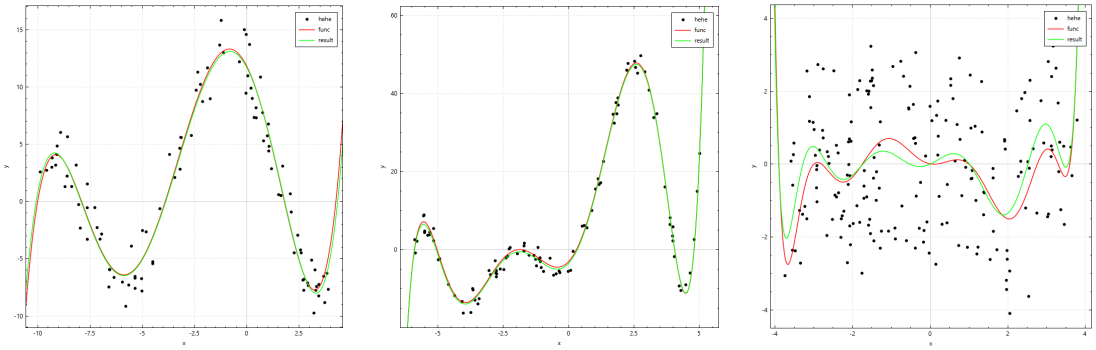
```
static void fillMatrix(MatrixXd& mtr, int const x1, int const x2, int const y1, int const y2, auto const& f) {
    assert((x1 >= 0) && (y1 >= 0) && (x1 < x2) && (y1 < y2) && (x2 <= mtr.cols()) && (y2 <= mtr.rows()));
    for (int i = y1; i < y2; ++i) {
        for (int j = x1; j < x2; ++j) {
            mtr(i, j) = f(i, j);
        }
    }
}

static qreal norm(auto const& vec) {
    qreal result = 0;
    for (auto const& elem : vec) {
        result += elem * elem;
    }
    return std::sqrt(result);
}

pr<VectorXd, int> gaussNewtonPolyRegression(const v<pr<qreal, qreal> > &points, const int max_step, const qreal dlt, int poly_deg)
{

    MatrixXd mtr(points.size(), poly_deg+1);
    fillMatrix(mtr, 0, 1, 0, mtr.rows(), [](int i, int j) { return 1; });
    fillMatrix(mtr, 1, mtr.cols(), 0, mtr.rows(), [&points](int i, int j) { return std::pow(points[i].first, j); });
    MatrixXd trans_mtr = mtr.transpose();
    VectorXd result(poly_deg+1);
    VectorXd diff(points.size());
    qreal scalar;
    MatrixXd tmp(poly_deg + 1, poly_deg + 1);
    VectorXd delta;

    for (int i = 0; i < max_step; ++i) {
        for (int j = 0; j < points.size(); ++j) {
            scalar = 0;
            for (int k = 0; k <= poly_deg; ++k) scalar += result[k] * mtr(j, k);
            diff[j] = points[j].second - scalar;
        }
        tmp = trans_mtr * mtr + MatrixXd::Identity(poly_deg + 1, poly_deg + 1);
        delta = tmp.colPivHouseholderQr().solve(trans_mtr * diff);
        result += delta;
#if DEBUG_OUTPUT
        qDebug() << norm(delta);
#endif
        if (norm(delta) < dlt) {
            return {std::move(result), i};
        }
    }
    return {std::move(result), max_step};
}
```

Описание кода:

Первая функция берет матрицу и диапазон по которому нужно пройтись и записывает туда значени возвращаемые функцией `f`.

Следуюая функция возвращает L2-норму по вектору. Другими словами его длину. Необходимо для остановки алгоритма Гаусса-Ньютона при смещении вектора полинома на зачение меньшее чем `dlt`.

Последняя функция - алгоритм Гаусса-Ньютона. Принимает множество точек, по которым строит кривую, максимальное количество шагов, эпсилон и разряд воззвращаемого полинома.



(a) Степень 5 (b) Степень 7 (c) Степень 10

Метод Гаусса-Ньютона обрабатывает сложные нелинейные модели, учитывает нелинейные эффекты и находит оптимальные значения коэффициентов. В отличие от классического метода наименьших квадратов (МНК), он обладает лучшей сходимостью при использовании больших полиномов. Однако метод Гаусса-Ньютона может привести к эффекту переобучения, когда модель слишком точно подстраивается под обучающие данные и плохо обобщается на новые.

2.2 Powell Dog Leg

Метод Powell Dog Leg (также известный как метод Пауэлла с обрезанным шагом) - это итерационный численный метод для нахождения минимума функции. Он основан на комбинации двух других методов - метода Пауэлла и метода обрезанного шага.

Метод Пауэлла является итерационным методом, который последовательно двигается по линиям нескольких координат, чтобы найти направление наискорейшего убывания функции. Он эффективен для задач с ограниченной областью поиска.

Метод обрезанного шага, с другой стороны, позволяет оптимизировать функцию с использованием только ограниченного набора доступных шагов. Он основан

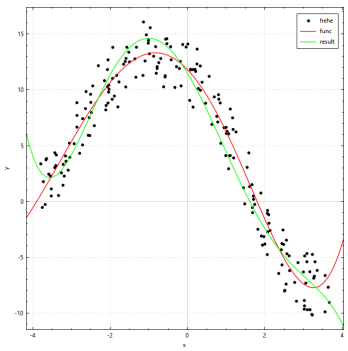
на том, что величина шага должна быть ограничена некоторым значением.

Метод Powell Dog Leg комбинирует эти два метода, чтобы найти оптимальное направление для движения в пространстве параметров функции. Он использует метод Пауэлла, когда функция не имеет ограничений, и метод обрезанного шага, когда необходимо учитывать ограничения.

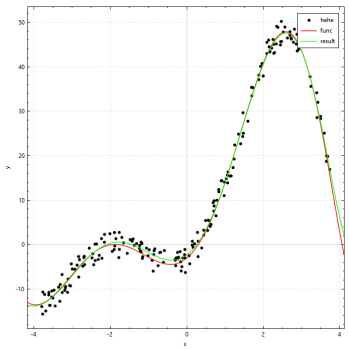
Таким образом, метод Powell Dog Leg представляет собой гибридный метод, который объединяет преимущества методов Пауэлла и обрезанного шага для эффективного поиска минимума функции.

Рис. 3. Код Powell Dog Leg

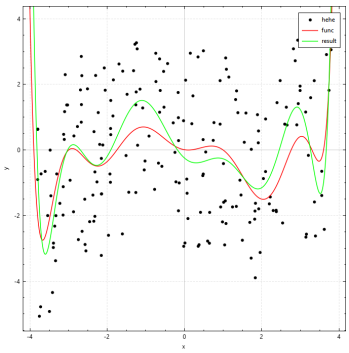
```cpp
static VectorXd powellDogLegGrad(const v<pr<qreal, qreal> > &points, MatrixXd const& mtr, MatrixXd const& trans_mtr, VectorXd const& cf) {
    VectorXd pred = mtr * cf;
    VectorXd diff(points.size(), 1);
    fillVector(diff, 0, points.size(), [&points, &pred](int i){ return pred[i] - points[i].second; });
    return trans_mtr * diff;
}

static VectorXd powellDogLegStep(VectorXd const& grad, MatrixXd const& hmtr, qreal const r) {
    VectorXd ustep = -grad / norm(grad);
    VectorXd bstep = hmtr.colPivHouseholderQr().solve(grad);
    if (norm(bstep) <= r) {
        return bstep;
    }
    qreal ub = ustep.dot(bstep);
    qreal uu = ustep.dot(ustep);
    qreal bb = bstep.dot(bstep);
    qreal res, temp = ub - uu + std::sqrt((ub - uu) * (ub - uu) + (r * r - uu) * bb);
    res = (temp == 0) ? 0 : (r * r - uu) / temp;
    return (1 - res) * bstep + res * ustep;
}

pr<VectorXd, int> powellDogLegRegression(const v<pr<qreal, qreal> > &points, const int max_step, const qreal dlt, int poly_deg)
{
    MatrixXd mtr(points.size(), poly_deg+1);
    fillMatrix(mtr, 0, 1, 0, mtr.rows(), [](int i, int j) { return 1; });
    fillMatrix(mtr, 1, mtr.cols(), 0, mtr.rows(), [&points](int i, int j) { return std::pow(points[i].first, j); });
    MatrixXd trans_mtr = mtr.transpose();
    VectorXd gs(poly_deg+1), cf(poly_deg+1);
    qreal r = 1, fig;
    VectorXd grad = powellDogLegGrad(points, mtr, trans_mtr, cf);
    MatrixXd hmtr = 2 * (trans_mtr * mtr);
    for (int i = 1; i <= max_step; ++i) {
        if (norm(grad) < dlt) {
            return {std::move(cf), i};
        }
        auto temp = powellDogLegStep(grad, hmtr, r);
        fig = std::pow(norm(temp), 2) / temp.dot(hmtr * temp);
        cf = cf + fig * temp;
        grad = powellDogLegGrad(points, mtr, trans_mtr, cf);
        hmtr = 2 * (trans_mtr * mtr);
    }
    return {std::move(cf), max_step};
}
```



(a) Степень 5



(b) Степень 7



(c) Степень 10

2.3 Статистика

Степень	Обычный SDG	Гаусс-Ньютон	Powell Dog Leg
1	19	10	60
2	39	13	67
3	59	14	59
4	78	16	61
5	117	20	61
6	137	21	63
7	156	26	63
8	176	27	64

2.4 Вывод по Gauss-Newton и Powell Dog Leg

Метод оптимизации Powell Dog Leg имеет несколько преимуществ. Во-первых, он способен работать с нелинейными ограничениями, что делает его гибким для различных типов задач. Во-вторых, он хорошо справляется с проблемами, связанными с наличием локальных минимумов или плато, что позволяет ему эффективно искать оптимальное решение.

Однако у метода Powell Dog Leg также есть некоторые недостатки. В частности, он может потребовать больше итераций для достижения сходимости по сравнению с другими методами оптимизации.

С другой стороны, метод оптимизации Gauss-Newton обладает своими преимуществами. Он особенно эффективен для решения задач нелинейных наименьших квадратов и быстро сходится в случае хорошо обусловленных задач с небольшим количеством переменных.

Однако у метода Gauss-Newton также есть свои недостатки. Во-первых, он может застревать в локальных минимумах, что может ограничивать его способность найти глобальный минимум. Во-вторых, для применения этого метода требуется вычисление и обращение матрицы Якобиана, что может быть вычислительно затратным и требовать большой объем памяти.

В целом, методы Powell Dog Leg и Gauss-Newton представляют собой различные подходы к оптимизации и имеют свои сильные и слабые стороны, которые могут быть учтены при выборе подходящего метода для конкретной задачи.

2.5 BFGS

Метод BFGS (Broyden-Fletcher-Goldfarb-Shanno) является одним из популярных методов оптимизации безусловной нелинейной оптимизации. Он относится к классу квазиньютоновских методов, которые используют аппроксимацию гессиана функции для обновления шага оптимизации. Основная идея метода BFGS заключается в последовательном уточнении приближения гессиана, чтобы найти оптимальное решение.

Основные формулы, используемые в методе BFGS, включают:

Обновление приближения гессиана: Гессиан H_{k+1} на k+1-м шаге вычисляется на основе предыдущего приближения H_k и информации о шаге оптимизации: $H_{k+1} = H_k + \Delta H$, где ΔH - поправка, которая вычисляется с использованием следующей формулы: $\Delta H = \rho \cdot u \cdot u^T - \rho \cdot H_k \cdot v \cdot v^T$, где $u = x_{k+1} - x_k$ и $v = \nabla f(x_k + 1) - \nabla f(x_k)$, ρ - множитель, который можно рассчитать на основе информации о шаге оптимизации и градиента функции.

Обновление шага оптимизации: Шаг оптимизации p_{k+1} на $k + 1$-м шаге вычисляется с использованием приближения гессиана и градиента функции: $p_{k+1} = -H_{k+1} \cdot \nabla f(x_{k+1})$, где $\nabla f(x_{k+1})$ - градиент функции на k+1-м шаге.

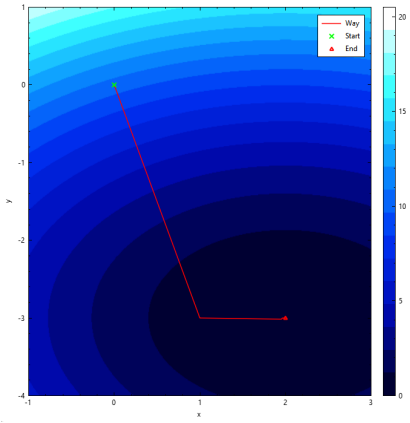
Обновление точки оптимизации: Точка оптимизации x_{k+1} вычисляется путем добавления шага оптимизации к текущей точке оптимизации: $x_{k+1} = x_k + p_{k+1}$.

Обновление приближения градиента: Приближение градиента гессиана G_{k+1} вычисляется на основе предыдущего приближения G_k и информации о шаге оптимизации: $G_{k+1} = G_k + \Delta G$, где ΔG - поправка, которая вычисляется с использованием следующей формулы: $\Delta G = \rho \cdot \nabla f(x_{k+1}) \cdot \nabla f(x_{k+1})^T - \rho \cdot G_k \cdot H_{k+1} \cdot G_k$.

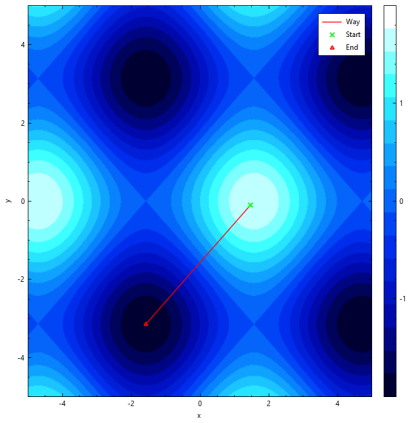
Метод BFGS обеспечивает быструю сходимость к оптимальному решению и обладает хорошей устойчивостью на практике. Он является итерационным методом, и его шаги обновления позволяют уточнить приближение гессиана и точку оптимизации, чтобы приблизиться к глобальному минимуму функции.

Рис. 5. Код BFGS

```cpp
static qreal BFGSStep(auto const& f, auto const& grd, VectorXd const& x, VectorXd const& step) {
    qreal lr = 1;
    while ((f(x + lr * step) > f(x) + 0.5 * lr * grd(x).dot(step))/*
        || (grd(x + lr * step).dot(step) < 0.9 * grd(x).dot(step))*/) {
        lr /= 2;
    }
    return lr;
}


v<QCPCurveData> BFGS(auto const& f, auto const& grd, VectorXd cur, const int max_step, const qreal dlt)
{
    auto norm = [](auto const& vec) {
        qreal result = 0;
        for (auto const& elem : vec) {
            result += elem * elem;
        }
        return std::sqrt(result);
    };
    int const n = cur.size();
    MatrixXd mtr = MatrixXd::Identity(n, n);
    VectorXd grad, step, next, diff, sy;
    v<QCPCurveData> way = {{-1, cur[0], cur[1]}};
    qreal lr;

    for (int i = 0; i < max_step; ++i) {
        grad = grd(cur);
        if (norm(grad) < dlt) {
            break;
        }
        step = (mtr * grad) * -1;
        lr = BFGSStep(f, grd, cur, step);
        next = cur + lr * step;
        diff = next - cur;
        sy = grd(next) - grad;
        if (sy.dot(diff) > 0) {
            VectorXd mtr_s = mtr * diff;
            qreal s_mtr_s = diff.dot(mtr_s);
            mtr += (sy * sy.transpose()) / sy.dot(diff) - (mtr_s * mtr_s.transpose()) / s_mtr_s;
        }
        cur = next;
        way.emplace_back(i, cur[0], cur[1]);
    }
    return way;
}
```



(a) $f(x, y) = \frac{1}{2}(x-2)^2 + (x+3)^2$. Шагов 6



(b) $f(x, y) = \sin(x) + \cos(y)$.Шагов 10

BFGS - эффективный алгоритм оптимизации с быстрой сходимостью к точке минимума. В сравнении с обычным градиентным спуском, BFGS требует гораздо меньше итераций (примерно в 3-4 раза меньше), чтобы достичь сходимости. Также, по сравнению с другими алгоритмами, такими как Powell Dog Leg и Gauss-Newton, BFGS имеет свои преимущества и недостатки.

Преимущества BFGS включают:

1. Быструю сходимость для широкого класса задач оптимизации.
2. Не требует вычисления и обращения матрицы Якобиана, а использует аппроксимацию гессиана на основе истории предыдущих итераций.

Однако, у BFGS есть и недостатки:

1. Большие затраты памяти для хранения информации о предыдущих итерациях. Для хранения псевдо-гессиана требуется O(n) памяти, где - n - количество переменных в задаче оптимизации.

2.6 L-BFGS

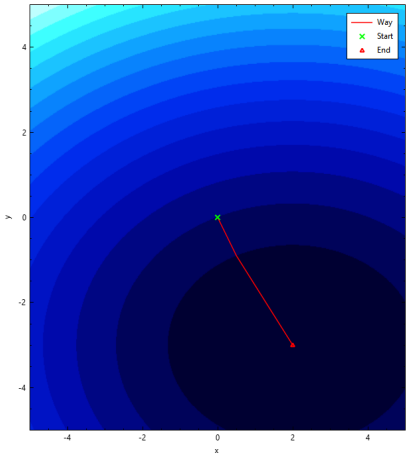
Метод L-BFGS (Limited-memory Broyden-Fletcher-Goldfarb-Shanno) - это оптимизационный алгоритм, используемый для решения задач безусловной оптимизации. Он является итерационным методом, который основывается на аппроксимации гессиана функции цели.

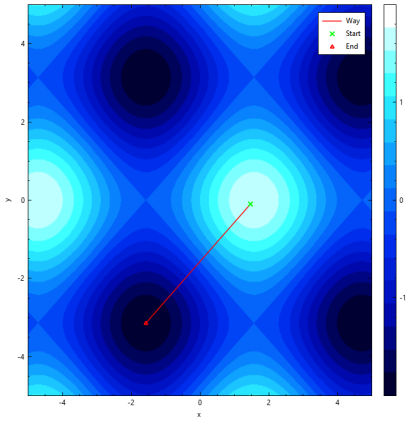
Основная идея метода L-BFGS заключается в аппроксимации гессиана функции цели с помощью некоторого компактного представления, использующего ограниченное количество предыдущих итераций. Это позволяет избежать хранения и обращения полного гессиана, что делает метод L-BFGS особенно эффективным для задач с большим числом переменных.

Метод L-BFGS хорошо работает, когда нужно решать сложные задачи оптимизации с большим количеством переменных. Он использует небольшое количество памяти для сохранения информации о предыдущих шагах, что помогает справиться с большими задачами. Сочетание информации о градиентах и значениях функции позволяет эффективно приближать гессиан функции и находить оптимальное решение. Однако стоит отметить, что иногда метод BFGS может быть более эффективным, особенно для задач с небольшим количеством переменных или когда задача хорошо условлена.

Рис. 7. Код BFGS

```cpp
static VectorXd LBFGSStep(VectorXd const& grad, que<VectorXd> const& s_hist, que<VectorXd> const& y_hist, que<qreal> const& rho_hist, MatrixXd const& H0) {
    VectorXd q = grad;
    v<qreal> alpha(s_hist.size());
    for (int i = s_hist.size() - 1; i >= 0; --i) {
        alpha[i] = rho_hist[i] * s_hist[i].dot(q);
        fillVector(q, 0, q.size(), [&q, &alpha, &y_hist, &i](int j) { return q[j] - alpha[i] * y_hist[i][j]; });
    }
    VectorXd r = H0 * q;
    qreal beta;
    for (int i = 0; i < s_hist.size(); ++i) {
        beta = rho_hist[i] * y_hist[i].dot(r);
        fillVector(r, 0, r.size(), [&i, &alpha, &beta, &s_hist, &r](int j){ return r[j] + s_hist[i][j] * (alpha[i] - beta);});
    }
    return -r;
}

static MatrixXd updtH(MatrixXd const& H0, que<VectorXd> const& s_hist, que<VectorXd> const& y_hist) {
    qreal q = y_hist.back().dot(s_hist.back()) / (y_hist.back().dot(y_hist.back()) + 1.e-8);
    qreal r = 1.0 / (y_hist.back().dot(s_hist.back()) + 1.e-4);
    return H0 + (q * y_hist.back() * y_hist.back().transpose()) - (r * s_hist.back() * s_hist.back().transpose());
}


v<QCPCurveData> LBFGS(auto const& f, auto const& grd, VectorXd x0, const int max_step, const qreal dlt, int const m)
{
    auto norm = [](auto const& vec) {
        qreal result = 0;
        for (auto const& elem : vec) {
            result += elem * elem;
        }
        return std::sqrt(result);
    };
    int const n = x0.size();
    VectorXd x = x0;
    VectorXd grad = grd(x);
    MatrixXd H0 = MatrixXd::Identity(n, n);
    que<VectorXd> s_hist, y_hist;
    que<qreal> rho_hist;
    v<QCPCurveData> way = {{-1, x[0], x[1]}};
    for (int i = 0; i < max_step; ++i) {
        VectorXd p = LBFGSStep(grad, s_hist, y_hist, rho_hist, H0);
        VectorXd s = 0.1 * p;
        VectorXd x_new = x + s;
        VectorXd g_new = grd(x_new);
        if (norm(g_new) < dlt) {
            break;
        }
        VectorXd y = g_new - grad;
        qreal rho = 1. / (y.dot(s) + 1.e-4);
        if (rho > 0) {
            s_hist.push_back(s);
            y_hist.push_back(y);
            rho_hist.push_back(rho);
            if (s_hist.size() > m) {
                s_hist.pop_front();
                y_hist.pop_front();
                rho_hist.pop_front();
            }
        }
        H0 = updtH(H0, s_hist, y_hist);
        x = x_new;
        grad = g_new;
        way.emplace_back(i, x[0], x[1]);
    }
    return way;
}
```





(a) $f(x, y) = \frac{1}{2}(x-2)^2 + (x+3)^2$. Шагов 6 (b) $f(x, y) = \sin(x) + \cos(y)$.Шагов 10

3 Вывод

Для успешного выполнения данной работы мы полностью реализовали все необходимые методы. После этого мы провели тщательное измерение производительности этих методов и осуществили детальный сравнительный анализ.