МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

Кафедра программных систем и баз данных

Лабораторная работа №3

по дисциплине оценивание параметров в обратных задачах

Факультет: ФПМИ

Группа: ПМ-63

Студенты: Крашенинник Н.А.

Пешкичева А.А.

Шепрут И.И.

Преподаватель: Вагин Д.В.

Новосибирск

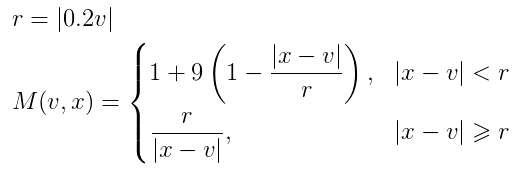
2019

1. **Задание**

Реализовать ПГА. Генотип состоит из N чисел. Фенотип состоит из M чисел. Числа фенотипа вычисляются регулярным образом из чисел генотипа.

С помощью ПГА восстановить коэффициенты полинома десятой степени по его значениям в заданном наборе точек. Значения зашумить на 1, 2, 5%.

1. **Метрика для алгоритма**



Здесь V – истинное значение по оси Y точки, Х – значение, предсказанное текущим полиномом. М – функция метрики. Генетический алгоритм старается максимизировать среднее этой функции для всех точек. Её максимально значение равно 10, минимальное 0.

1. **Примеры нахождения решений**

Шум – 0%, поколений – 200, размер популяции – 30, количество точек – 15.

Metric for true coefficients: 10.000

Metric for result: 9.745

Distance: 54.560

Found True |f - t|

------------------------------

65.779 -10.000 75.779

-75.076 0.000 75.076

-69.707 3.100 72.807

152.452 88.000 64.452

4.603 5.000 0.397

-16.845 -15.000 1.845

9.997 10.000 0.003

Шум – 0%, поколений – 2000, размер популяции – 60, количество точек – 30.

Metric for true coefficients: 10.000

Metric for result: 9.996

Distance: 8.187

Found True |f - t|

------------------------------

4.449 -10.000 14.449

-15.732 0.000 15.732

6.695 3.100 3.595

88.071 88.000 0.071

5.123 5.000 0.123

-15.007 -15.000 0.007

9.998 10.000 0.002

Шум – 0%, поколений – 2000, размер популяции – 60, количество точек – 30.

Metric for true coefficients: 9.419

Metric for result: 9.440

Distance: 0.525

Found True |f - t|

------------------------------

-10.360 -10.000 0.360

0.748 0.000 0.748

3.809 3.100 0.709

88.224 88.000 0.224

5.743 5.000 0.743

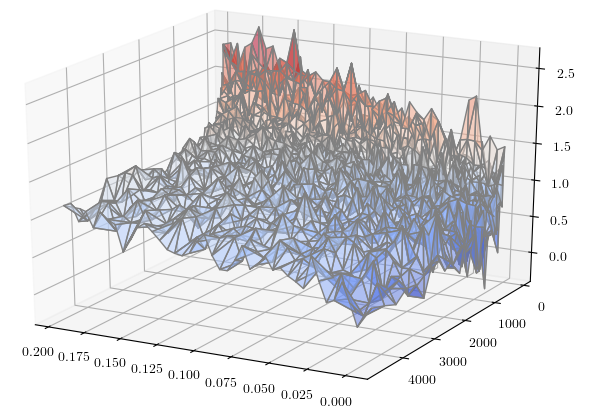
-14.636 -15.000 0.364

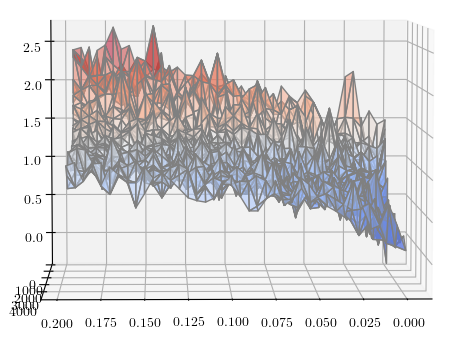
10.006 10.000 0.006

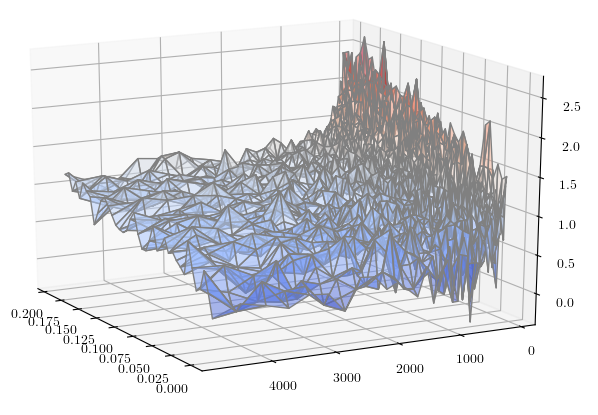
1. **График**

По одной горизонтальной оси показано число точек, которые использовалсь для предсказания коэффициентов полинома. По другой горизонтальной оси показан процент равномерного шума, добавленный к каждой точке. По вертикальной оси показано евклидово расстояние между истинными коэффициентами и найденными.

Для построения этого графика использовалось очень много запусков генетического алгоритма с разной инициализацией генератора случайных чисел. Данный график рассчитывался 2 дня и 4 часа.







1. **Выводы**

* Чем больше процент шума, тем хуже находятся коэффициенты полинома.
* Чем больше количество точек, тем лучше находятся коэффициенты полинома.
* Генетический алгоритм сильно зависит от случайности.
* Если посмотреть на метрику истинного решения и метрику найденного в 3.3, то поймём, что метрика истинного меньше, чем найденного, это значит, что максиимизируя её, мы лишь стремимся к решению, но не значит что если мы найдём её максимум, мы получим решение с максимальной точностью. В машинном обучении это называется «переобучение».

1. **Код**

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <random>

#include <memory>

#include <vector>

#include <string>

#include <iostream>

#include <iomanip>

#include <functional>

#include <fstream>

#include <thread>

#include <future>

#include <map>

#include <mutex>

#include <chrono>

#include <sstream>

class PolynomCreature;

typedef std::shared\_ptr<PolynomCreature> PolynomCreature\_ptr;

class RandomGenerator {

public:

RandomGenerator(int seed) : generator(seed) {

}

double randomDouble(double a, double b) {

std::uniform\_real\_distribution<double> distribution(a, b);

return distribution(generator);

}

int randomInt(int a, int b) {

std::uniform\_int\_distribution<int> distribution(a, b);

return distribution(generator);

}

private:

std::mt19937 generator;

};

RandomGenerator rnd(0);

class PolynomCreature {

public:

void mutateSingle(void) {

values[rnd.randomInt(0, values.size()-1)] \*= rnd.randomDouble(-2, 2);

}

void mutateSlightly(void) {

values[rnd.randomInt(0, values.size() - 1)] \*= rnd.randomDouble(0.9, 1.1);

}

void mutateSuperSlightly(void) {

values[rnd.randomInt(0, values.size() - 1)] \*= rnd.randomDouble(0.91, 1.01);

}

PolynomCreature() = default;

// Конструктор копирования, нужен для создания неизменных особей

PolynomCreature(const PolynomCreature& other) = default;

double calcPolynomInPoint(double x) const {

double sum = 0;

double powx = 1;

for (auto& i : values) {

sum += i \* powx;

powx \*= x;

}

return sum;

}

std::vector<double> values;

};

PolynomCreature\_ptr makeRandom(int size, double a, double b) {

PolynomCreature result;

result.values.resize(size);

for (auto& i : result.values) {

i = rnd.randomDouble(a, b);

}

return PolynomCreature\_ptr(new PolynomCreature(result));

}

double evalByPosition(double value, double sum, int pos, int size) {

return 1 / pow(1.1, pos+1);

}

double evalByValue(double value, double sum, int pos, int size) {

return value;

}

template<class Creature>

class Evolution {

public:

typedef std::shared\_ptr<Creature> Creature\_ptr;

typedef std::vector<Creature\_ptr> Creatures;

typedef std::pair<Creature\_ptr, double> CreatureWithScore;

typedef std::vector<CreatureWithScore> CreatureScores;

typedef std::function<Creature\_ptr(void)> CreatureGenerator;

typedef std::function<double(const Creature&)> EvalFunction;

// Maximizes

Evolution(

int populationSize,

EvalFunction max,

CreatureGenerator generator

) : max(max), generator(generator), mutateNormal(true), mutateSuperSlightly(false) {

init(populationSize);

}

void init(int populationSize) {

population.clear();

population.reserve(populationSize);

for (int i = 0; i < populationSize; ++i) {

population.push\_back(generator());

}

}

CreatureWithScore evolve(int attempts, bool isPrint) {

CreatureWithScore best = {population[0], max(\*population[0])};

for (int i = 0; i < attempts; ++i) {

init(population.size());

auto res = evolveAttempt();

if (res.second > best.second) {

best = res;

if (isPrint) {

std::cout << "\r" << "Found best creature with metric: " << std::setprecision(3) << best.second << std::endl;

}

}

if (i % 10 == 0 && isPrint) {

std::cout << "\r" << std::setw(10) << std::fixed << std::setprecision(1) << i \* 100.0 / attempts << '%';

}

}

if (isPrint)

std::cout << "\r" << std::setw(10) << " \r";

if (isPrint)

std::cout << "Start to optimize best creature" << std::endl;

mutateNormal = false;

mutateSuperSlightly = true;

std::vector<Creature\_ptr> population1;

for (auto& i : population)

population1.emplace\_back(new Creature(\*best.first));

population = population1;

for (int i = 0; i < attempts \* 10; ++i) {

auto scores = evalScores(evalByValue);

if (scores[0].second > best.second) {

best = scores[0];

if (isPrint) {

std::cout << "\r" << "Found best creature with metric: " << std::setprecision(3) << best.second << std::endl;

}

}

auto [population1, efficiency] = makeNewPopulation(scores);

population = population1;

}

mutateSuperSlightly = false;

return best;

}

private:

Creatures population;

EvalFunction max;

CreatureGenerator generator;

bool mutateNormal;

bool mutateSuperSlightly;

CreatureScores evalScores(const std::function<double(double value, double sum, int pos, int size)>& normalize) const {

CreatureScores result;

double sum = 0;

// Оцениваем каждое существо

for (const auto& i : population) {

double score = max(\*i);

result.push\_back({i, score});

sum += score;

}

// Сортируем существ по их оценкам

std::sort(result.begin(), result.end(), [] (const auto& a, const auto& b) -> bool {

return a.second > b.second;

});

// Применяем к числам определённую нормализацию

int pos = 0;

sum = 0;

for (auto& i : result) {

i.second = normalize(i.second, sum, pos, result.size());

sum += i.second;

pos++;

}

// Нормализуем все оценки таким образом, чтобы их сумма была равна 1

for (auto& i : result)

i.second /= sum;

return result;

}

std::pair<Creatures, double> makeNewPopulation(const CreatureScores& scores) {

Creatures result;

result.reserve(scores.size());

auto best = scores[0].first;

auto bestValue = scores[0].second;

result.emplace\_back(new Creature(\*best));

for (int i = 0; i < population.size()-1; ++i) {

result.emplace\_back(new Creature(\*best));

if (mutateSuperSlightly) {

if (i % 2 == 0) {

result.back()->mutateSingle();

result.back()->mutateSingle();

} else {

result.back()->mutateSuperSlightly();

}

} else {

if (mutateNormal) {

result.back()->mutateSingle();

} else {

result.back()->mutateSlightly();

}

}

}

return {result, max(\*best)};

}

CreatureWithScore evolveAttempt(void) {

mutateNormal = true;

std::vector<double> lge; // Last generations efficiency

const int exitCount = 20;

for (int i = 0; i < 2000; i++) {

auto scores = evalScores(evalByValue);

if (lge.size() == exitCount && lge.front() == lge.back()) {

return { scores[0].first, max(\*scores[0].first) };

}

auto [population1, efficiency] = makeNewPopulation(scores);

population = population1;

if (!lge.empty() && lge.back() == efficiency) {

mutateNormal = false;

}

lge.push\_back(efficiency);

if (lge.size() > exitCount) {

lge.erase(lge.begin());

}

}

auto scores = evalScores(evalByValue);

return { scores[0].first, max(\*scores[0].first) };

}

};

typedef std::vector<std::pair<double, double>> Points;

double distanceAbsolute(double a, double b) {

return pow(a - b, 2);

}

double distance(const PolynomCreature& coefs1, const PolynomCreature& coefs2) {

double sum = 0;

for (int i = 0; i < coefs1.values.size(); i++) {

sum += distanceAbsolute(coefs1.values[i], coefs2.values[i]);

}

return sqrt(sum/coefs1.values.size());

}

double calcUniformMetric(double value, double x, double percent) {

double r = fabs(value \* percent \* 2);

if (fabs(x - value) < r) {

return 1 + (1 - fabs(x - value)/r) \* 9;

} else {

return r / fabs(x - value);

}

}

double distance(const Points& a, const PolynomCreature& c) {

double percent = 0.2;

double sum = 0;

for (int i = 0; i < a.size(); i++) {

sum += calcUniformMetric(a[i].second, c.calcPolynomInPoint(a[i].first), percent);

}

return sum / a.size();

}

double calcEvolutionDistance(const PolynomCreature& creature, int generations, int populationSize, int pointsCount, double noisePercent, bool isPrint) {

Points points;

for (int i = 0; i < pointsCount; ++i) {

double x = rnd.randomDouble(-20, 20);

double y = creature.calcPolynomInPoint(x);

y \*= 1 + rnd.randomDouble(-noisePercent, noisePercent);

points.push\_back({x, y});

}

Evolution<PolynomCreature> e(populationSize, [&](const PolynomCreature& c) -> double {

return distance(points, c);

}, [&]() -> PolynomCreature\_ptr {

return makeRandom(creature.values.size(), -100, 100);

});

auto res = e.evolve(generations, isPrint);

using namespace std;

if (isPrint) {

cout << setprecision(3);

cout << "Metric for true coefficients: " << distance(points, creature) << endl;

cout << "Metric for result: " << res.second << endl;

cout << "Distance: " << distance(creature, \*res.first) << endl;

cout << endl;

cout << setprecision(3);

cout << setw(10) << "Found" << setw(10) << "True" << setw(10) << "|f - t|" << endl;

cout << setw(10) << "----------" << setw(10) << "----------" << setw(10) << "----------" << endl;

auto& coefs = creature.values;

auto& result = res.first->values;

for (int i = 0; i < coefs.size(); i++) {

cout << setw(10) << result[i] << setw(10) << coefs[i] << setw(10) << fabs(result[i] - coefs[i]) << endl;

}

cout << endl;

}

return distance(creature, \*res.first);

}

class percent\_time\_analyzer

{

public:

percent\_time\_analyzer() : time(0) {}

void start() {

using namespace std;

time = getCurrentTime();

cout << setfill(' ') << setiosflags(ios\_base::right);

cout << "Percent |" << setw(23) << "Time passed |" << setw(23) << "Approximate time |" << setw(24) << "Time Left |" << endl;

cout << setfill('-') << setw(79) << '|' << endl;

cout << setfill(' ');

}

void print\_percent(double percent) {

static std::mutex m;

std::lock\_guard<std::mutex> g(m);

using namespace std;

stringstream sout;

//cout << '\r';

sout.str(std::string());

sout << setprecision(2) << percent \* 100 << "% |";

cout << setw(9) << sout.str();

sout.str(std::string());

sout << getTimeString(getTimePassed(time)) << " |";

cout << setw(23) << sout.str();

sout.str(std::string());

sout << getTimeString(getApproxTime(time, percent)) << " |";

cout << setw(23) << sout.str();

sout.str(std::string());

sout << getTimeString(getLeftTime(time, percent)) << " |";

cout << setw(24) << sout.str();

cout << endl;

}

void end() {

using namespace std;

cout << '\r' << setw(9) << "100% |";

stringstream sout;

sout.clear();

sout << getTimeString(getTimePassed(time)) << " |";

cout << setw(23) << sout.str();

cout << setw(23) << "0s |" << setw(24) << "0s |" << endl;

cout << endl;

}

private:

double time;

typedef std::chrono::high\_resolution\_clock hrc;

//-------------------------------------------------------------------------

float getCurrentTime(void) {

static hrc::time\_point t = hrc::now();

return std::chrono::duration<double>(hrc::now() - t).count();

}

//-------------------------------------------------------------------------

float getTimePassed(float pastTime) {

return getCurrentTime() - pastTime;

}

//-------------------------------------------------------------------------

float getApproxTime(float pastTime, float percent) {

if (percent == 0)

return 0;

else

return getTimePassed(pastTime) / percent;

}

//-------------------------------------------------------------------------

float getLeftTime(float pastTime, float percent) {

if (percent == 0)

return 0;

else

return getApproxTime(pastTime, percent) - getTimePassed(pastTime);

}

//-------------------------------------------------------------------------

std::string getTimeString(float time) {

char s[25] = {};

if (true) {

if (time > 86400)

sprintf(s, "%2dd %2dh %2dm %2ds", int(time / 86400), int(time / 3600) % 24, int(time / 60) % 60, int(time) % 60);

else

if (time > 3600)

sprintf(s, " %2dh %2dm %2ds", int(time / 3600) % 24, int(time / 60) % 60, int(time) % 60);

else

if (time > 60)

sprintf(s, " %2dm %2ds", int(time / 60) % 60, int(time) % 60);

else

sprintf(s, " %2ds", int(time) % 60);

}

else {

if (time > 86400)

sprintf(s, "%2dd %2dh %2dm %2ds", int(time / 86400), int(time / 3600) % 24, int(time / 60) % 60, int(time) % 60);

else

if (time > 3600)

sprintf(s, "%2dh %2dm %2ds", int(time / 3600) % 24, int(time / 60) % 60, int(time) % 60);

else

if (time > 60)

sprintf(s, "%2dm %2ds", int(time / 60) % 60, int(time) % 60);

else

sprintf(s, "%2ds", int(time) % 60);

}

return std::string(s);

}

};

template<class Ret, class Key>

class async\_performer\_t

{

public:

void add(const std::function<Ret(void)>& f, const Key& key) {

mf.push\_back({key, f});

}

void finish(void) {

int counter = 0;

std::mt19937 gen(0);

std::shuffle(mf.begin(), mf.end(), gen);

percent\_time\_analyzer time;

time.start();

int threads = 4;

std::mutex mm;

std::vector<std::thread> threads\_mas;

auto start = mf.begin();

auto count = mf.size() / threads;

for (int i = 0; i < threads; ++i) {

threads\_mas.push\_back(std::thread([&](typename MF::iterator start, typename MF::iterator end) {

for (auto i = start; i != end; ++i) {

auto value = i->second();

mm.lock();

m[i->first] = value;

counter++;

time.print\_percent(double(counter) / mf.size());

mm.unlock();

}

}, start, start + count));

start += count;

}

for (auto& i : threads\_mas)

i.join();

std::cout << "\r \r";

}

auto begin(void) { return m.begin(); }

auto end(void) { return m.end(); }

Ret& operator[](const Key& key) { return m[key]; }

const Ret& operator[](const Key& key) const { return m.at(key); }

private:

typedef std::vector<std::pair<Key, std::function<Ret(void)>>> MF;

MF mf;

std::map<Key, Ret> m;

};

int main() {

PolynomCreature creature;

creature.values = {-10, 0, 3.1, 88, 5, -15, 10};

auto result = calcEvolutionDistance(creature, 200, 30, 15, 0.0, true);

//result = calcEvolutionDistance(creature, 2000, 60, 30, 0.0, true);

result = calcEvolutionDistance(creature, 200, 30, 200, 0.05, true);

return 0;

double max\_points = 5000;

double start\_points = 8;

double mul\_points = 1.1;

double start\_percent = 0;

double max\_percent = 0.20;

double sum\_percent = 0.005;

int evolutionRepeats = 25;

int attempts = 50;

int populationSize = 30;

async\_performer\_t<double, std::pair<double, int>> performer;

int counter = 0;

int count = 0;

for (double percent = start\_percent; percent <= max\_percent; percent += sum\_percent) {

for (double points = start\_points; points < max\_points; points \*= mul\_points) {

count++;

}

}

for (double percent = start\_percent; percent <= max\_percent; percent += sum\_percent) {

for (double points = start\_points; points < max\_points; points \*= mul\_points) {

performer.add([percent, points, creature, attempts, populationSize, evolutionRepeats]() -> double {

double sum = 0;

for (int i = 0; i < evolutionRepeats; i++) {

auto result = calcEvolutionDistance(creature, attempts, populationSize, points, percent, false);

sum += result;

}

sum /= evolutionRepeats;

return sum;

}, {percent, points});

}

}

performer.finish();

std::ofstream fout("out.txt");

for (auto& i : performer) {

fout << i.first.first << "\t" << i.first.second << "\t" << i.second << std::endl;

}

fout.close();

}