```
/* Copyright (C) Jelle Spijker - All Rights Reserved
 * Unauthorized copying of this file, via any medium is strictly prohibited
 * and only allowed with the written consent of the author (Jelle Spijker)
 * This software is proprietary and confidential
 * Written by Jelle Spijker <spijker.jelle@gmail.com>, 2015
 */

#pragma once

#define GENE_MAX 32         /**< maximum number of genes*/
#define CROSSOVER 16        /**< crossover location*/
#define MUTATIONRATE 0.075f /**< mutation rate*/
#define ELITISME 4          /**< total number of the elite bastard*/
#define END_ERROR 0.005f    /**< acceptable error between last itteration*/

#include <stdint.h>
#include <bitset>
#include <vector>
#include <complex>
#include <valarray>
#include <array>

typedef unsigned char uchar;   /**< unsigned char*/
typedef unsigned short ushort; /**< unsigned short*/

typedef std::complex<double> Complex_t;       /**< complex vector of doubles*/
typedef std::vector<Complex_t> ComplexVect_t; /**< vector of Complex_t*/
typedef std::valarray<Complex_t> ComplexArray_t; /**< valarray of Complex_t*/
typedef std::vector<uint32_t> iContour_t;        /**< vector of uint32_t*/

typedef std::bitset<GENE_MAX> Genome_t; /**< Bitset repressenting a genome*/
typedef std::pair<std::bitset<CROSSOVER>, std::bitset<GENE_MAX - CROSSOVER>>
    SplitGenome_t;                          /**< a matted genome*/
typedef std::vector<float> Weight_t;     /**< a float vector*/
typedef std::vector<Genome_t> GenVect_t; /**< a vector of genomes*/
typedef struct PopMemberStruct {
  Weight_t weights;       /**< the weights the core of a population member*/
  GenVect_t weightsGen;   /**< the weights as genomes*/
  float Calculated = 0.0; /**< the calculated value*/
  float Fitness = 0.0;    /**< the fitness of the population member*/
} PopMember_t;            /**< a population member*/
typedef std::vector<PopMember_t> Population_t; /**< Vector with PopMember_t*/
typedef std::pair<float, float>
    MinMaxWeight_t; /**< floating pair weight range*/

typedef struct Predict_struct {
  uint32_t Category; /**< the category number */
  float RealValue;   /**< category number as float in order to estimate how
                        precise to outcome is*/
  float Accuracy;    /**< the accuracy of the category*/
  std::vector<float> OutputNeurons; /**< the output Neurons*/
} Predict_t;                        /**< The prediction results*/
typedef Predict_t (*NNfunctionType)(
    ComplexVect_t, Weight_t, Weight_t, uint32_t, uint32_t,
    uint32_t); /**< The prediction function from the Neural Net*/

typedef std::vector<ComplexVect_t>
    InputLearnVector_t; /**< Vector of a vector with complex values*/
typedef std::vector<Predict_t> OutputLearnVector_t; /**< vector with results*/
```

```
    1: /* Copyright (C) Jelle Spijker - All Rights Reserved
    2:  * Unauthorized copying of this file, via any medium is strictly prohibited
    3:  * and only allowed with the written consent of the author (Jelle Spijker)
    4:  * This software is proprietary and confidential
    5:  * Written by Jelle Spijker <spijker.jelle@gmail.com>, 2015
    6:  */
    7:
    8: #pragma once
    9: #define MAX_UINT8_VALUE 256
   10: #define VECTOR_CALC 1
   11: #define STATS_VERSION 1
   12:
   13: #include <stdint.h>
   14: #include <utility>
   15: #include <vector>
   16: #include <cstdlib>
   17: #include <cmath>
   18: #include <limits>
   19: #include <typeinfo>
   20: #include <string>
   21:
   22: #include <fstream>
   23:
   24: #include <boost/archive/binary_iarchive.hpp>
   25: #include <boost/archive/binary_oarchive.hpp>
   26: #include <boost/math/distributions/students_t.hpp>
   27:
   28: #include "MathException.h"
   29: #include "SoilMathTypes.h"
   30: #include "CommonOperations.h"
   31:
   32: using namespace std;
   33:
   34: namespace SoilMath {
   35:
   36: /*!
   37:  * \brief Stats class
   38:  * \details Usage Stats<type1, type2, type3>Stats() type 1, 2 and 3 shoudl be of
   39:  * the same value and concecuative in size
   40:  */
   41: template <typename T1, typename T2, typename T3> class Stats {
   42: public:
   43:   bool isDiscrete = true; /**< indicates if the data is discrete or real*/
   44:
   45:   T1 *Data;               /**< Pointer the data*/
   46:   uint32_t *bins;         /**< the histogram*/
   47:   bool Calculated = false; /**< indication if the data has been calculated*/
   48:   float Mean = 0.0;       /**< the mean value of the data*/
   49:   uint32_t n = 0;         /**< number of data points*/
   50:   uint32_t noBins = 0;    /**< number of bins*/
   51:   T1 Range = 0;           /**< range of the data*/
   52:   T1 min = 0;             /**< minimum value*/
   53:   T1 max = 0;             /**< maximum value*/
   54:   T1 Startbin = 0;        /**< First bin value*/
   55:   T1 EndBin = 0;          /**< End bin value*/
   56:   T1 binRange = 0;        /**< the range of a single bin*/
   57:   float Std = 0.0;        /**< standard deviation*/
   58:   T3 Sum = 0;             /**< total sum of all the data values*/
   59:   uint16_t Rows = 0;      /**< number of rows from the data matrix*/
   60:   uint16_t Cols = 0;      /**< number of cols from the data matrix*/
   61:   bool StartAtZero = true; /**< indication of the minimum value starts at zero
   62:                               or could be less*/
   63:
   64:   uint32_t *begin() { return &bins[0]; }    /**< pointer to the first bin*/
   65:   uint32_t *end() { return &bins[noBins]; } /**< pointer to the last + 1 bin*/
   66:
   67:   /*!
   68:    * \brief WelchTest Compare the sample using the Welch's Test
   69:    * \details (source:
   70:    * http://www.boost.org/doc/libs/1_57_0/libs/math/doc/html/math_toolkit/stat_tut/weg/st_eg/two_sample_stu
dents_t.html)
   71:    * \param statComp Statiscs Results of which it should be tested against
   72:    * \return
   73:    */
   74:   bool WelchTest(SoilMath::Stats<T1, T2, T3> &statComp) {
   75:     double alpha = 0.05;
   76:     // Degrees of freedom:
   77:     double v = statComp.Std * statComp.Std / statComp.n +
   78:                this->Std * this->Std / this->n;
   79:     v *= v;
   80:     double t1 = statComp.Std * statComp.Std / statComp.n;
   81:     t1 *= t1;
   82:     t1 /= (statComp.n - 1);
```

```
   83:     double t2 = this->Std * this->Std / this->n;
   84:     t2 *= t2;
   85:     t2 /= (this->n - 1);
   86:     v /= (t1 + t2);
   87:     // t-statistic:
   88:     double t_stat = (statComp.Mean - this->Mean) /
   89:                     sqrt(statComp.Std * statComp.Std / statComp.n +
   90:                          this->Std * this->Std / this->n);
   91:     //
   92:     // Define our distribution, and get the probability:
   93:     //
   94:     boost::math::students_t dist(v);
   95:     double q = cdf(complement(dist, fabs(t_stat)));
   96:
   97:     bool rejected = false;
   98:     // Sample 1 Mean == Sample 2 Mean test the NULL hypothesis, the two means
   99:     // are the same
  100:     if (q < alpha / 2)
  101:       rejected = false;
  102:     else
  103:       rejected = true;
  104:     return rejected;
  105:   }
  106:
  107:   /*!
  108:    * \brief Stats Constructor
  109:    * \param rhs Right hand side
  110:    */
  111:   Stats(const Stats &rhs)
  112:       : Data{new T1[rhs.n]}, bins{new uint32_t[rhs.noBins]} {
  113:     this->binRange = rhs.binRange;
  114:     this->Calculated = rhs.Calculated;
  115:     this->Cols = rhs.Cols;
  116:     this->EndBin = rhs.EndBin;
  117:     this->isDiscrete = rhs.isDiscrete;
  118:     this->max = rhs.max;
  119:     this->Mean = rhs.Mean;
  120:     this->min = rhs.min;
  121:     this->n = rhs.n;
  122:     this->noBins = rhs.noBins;
  123:     this->n_end = rhs.n_end;
  124:     this->Range = rhs.Range;
  125:     this->Rows = rhs.Rows;
  126:     this->Startbin = rhs.Startbin;
  127:     this->Std = rhs.Std;
  128:     this->Sum = rhs.Sum;
  129:     std::copy(rhs.bins, rhs.bins + rhs.noBins, this->bins);
  130:     this->Data = &rhs.Data[0];
  131:     this->StartAtZero = rhs.StartAtZero;
  132:   }
  133:
  134:   /*!
  135:    * \brief operator = Assigmnet operator
  136:    * \param rhs right hand side
  137:    * \return returns the right hand side
  138:    */
  139:   Stats &operator=(Stats const &rhs) {
  140:     if (&rhs != this) {
  141:       delete[] bins;
  142:       delete[] Data;
  143:       bins = new uint32_t[rhs.noBins];
  144:       Data = new T1[rhs.n];
  145:       this->binRange = rhs.binRange;
  146:       this->Calculated = rhs.Calculated;
  147:       this->Cols = rhs.Cols;
  148:       this->EndBin = rhs.EndBin;
  149:       this->isDiscrete = rhs.isDiscrete;
  150:       this->max = rhs.max;
  151:       this->Mean = rhs.Mean;
  152:       this->min = rhs.min;
  153:       this->n = rhs.n;
  154:       this->noBins = rhs.noBins;
  155:       this->n_end = rhs.n_end;
  156:       this->Range = rhs.Range;
  157:       this->Rows = rhs.Rows;
  158:       this->Startbin = rhs.Startbin;
  159:       this->Std = rhs.Std;
  160:       this->Sum = rhs.Sum;
  161:       this->Data = &rhs.Data[0];
  162:       std::copy(rhs.bins, rhs.bins + rhs.noBins, this->bins);
  163:       this->StartAtZero = rhs.StartAtZero;
  164:     }
  165:     return *this;
```

```
166:   }
167:
168:   /*!
169:    * \brief Stats Constructor
170:    * \param noBins number of bins with which to build the histogram
171:    * \param startBin starting value of the first bin
172:    * \param endBin end value of the second bin
173:    */
174:   Stats(int noBins = 256, T1 startBin = 0, T1 endBin = 255) {
175:     min = numeric_limits<T1>::max();
176:     max = numeric_limits<T1>::min();
177:     Range = numeric_limits<T1>::max();
178:     Startbin = startBin;
179:     EndBin = endBin;
180:     this->noBins = noBins;
181:     bins = new uint32_t[noBins]{};
182:
183:     if (typeid(T1) == typeid(float) || typeid(T1) == typeid(double) ||
184:         typeid(T1) == typeid(long double)) {
185:       isDiscrete = false;
186:       binRange = static_cast<T1>((EndBin - Startbin) / noBins);
187:     } else {
188:       isDiscrete = true;
189:       binRange = static_cast<T1>(round((EndBin - Startbin) / noBins));
190:     }
191:   }
192:
193:   /*!
194:    * \brief Stats constructor
195:    * \param data Pointer to the data
196:    * \param rows Number of rows
197:    * \param cols Number of Columns
198:    * \param noBins Number of bins
199:    * \param startBin Value of the start bin
200:    * \param startatzero bool indicating if the bins should be shifted from zero
201:    */
202:   Stats(T1 *data, uint16_t rows, uint16_t cols, int noBins = 256,
203:         T1 startBin = 0, bool startatzero = true) {
204:     min = numeric_limits<T1>::max();
205:     max = numeric_limits<T1>::min();
206:     Range = max - min;
207:
208:     Startbin = startBin;
209:     EndBin = startBin + noBins;
210:     StartAtZero = startatzero;
211:
212:     if (typeid(T1) == typeid(float) || typeid(T1) == typeid(double) ||
213:         typeid(T1) == typeid(long double)) {
214:       isDiscrete = false;
215:     } else {
216:       isDiscrete = true;
217:     }
218:
219:     Data = data;
220:     Rows = rows;
221:     Cols = cols;
222:     bins = new uint32_t[noBins]{};
223:     this->noBins = noBins;
224:     if (isDiscrete) {
225:       BasicCalculate();
226:     } else {
227:       BasicCalculateFloat();
228:     }
229:   }
230:
231:   /*!
232:    * \brief Stats Constructor
233:    * \param data Pointer the data
234:    * \param rows Number of rows
235:    * \param cols Number of Columns
236:    * \param mask the mask should have the same size as the data a value of zero
237:    * indicates that the data pointer doesn't exist. A 1 indicates that the data
238:    * pointer is to be used
239:    * \param noBins Number of bins
240:    * \param startBin Value of the start bin
241:    * \param startatzero indicating if the bins should be shifted from zero
242:    */
243:   Stats(T1 *data, uint16_t rows, uint16_t cols, uchar *mask, int noBins = 256,
244:         T1 startBin = 0, bool startatzero = true) {
245:     min = numeric_limits<T1>::max();
246:     max = numeric_limits<T1>::min();
247:     Range = max - min;
248:
```

```
249:     Startbin = startBin;
250:     EndBin = startBin + noBins;
251:     StartAtZero = startatzero;
252:
253:     if (typeid(T1) == typeid(float) || typeid(T1) == typeid(double) ||
254:         typeid(T1) == typeid(long double)) {
255:       isDiscrete = false;
256:     } else {
257:       isDiscrete = true;
258:     }
259:
260:     Data = data;
261:     Rows = rows;
262:     Cols = cols;
263:     bins = new uint32_t[noBins]{};
264:     this->noBins = noBins;
265:     if (isDiscrete) {
266:       BasicCalculate(mask);
267:     } else {
268:       BasicCalculateFloat(mask);
269:     }
270:   }
271:
272:   /*!
273:    * \brief Stats Constructor
274:    * \param binData The histogram data
275:    * \param startC start counter
276:    * \param endC end counter
277:    */
278:   Stats(T2 *binData, uint16_t startC, uint16_t endC) {
279:     noBins = endC - startC;
280:     Startbin = startC;
281:     EndBin = endC;
282:     uint32_t i = noBins;
283:
284:     if (typeid(T1) == typeid(float) || typeid(T1) == typeid(double) ||
285:         typeid(T1) == typeid(long double)) {
286:       isDiscrete = false;
287:       throw Exception::MathException(
288:           "Calculations using histogram not supported with floating-type!");
289:     } else {
290:       isDiscrete = true;
291:     }
292:
293:     bins = new uint32_t[noBins]{};
294:     while (i-- > 0) {
295:       bins[i] = binData[i];
296:       n += binData[i];
297:     }
298:     BinCalculations(startC, endC);
299:   }
300:
301:   ~Stats() { delete[] bins; }
302:
303:   /*!
304:    * \brief BasicCalculateFloat execute the basic float data calculations
305:    */
306:   void BasicCalculateFloat() {
307:     float sum_dev = 0.0;
308:     n = Rows * Cols;
309:     for (uint32_t i = 0; i < n; i++) {
310:       if (Data[i] > max) {
311:         max = Data[i];
312:       }
313:       if (Data[i] < min) {
314:         min = Data[i];
315:       }
316:       Sum += Data[i];
317:     }
318:     binRange = (max - min) / noBins;
319:     uint32_t index = 0;
320:     Mean = Sum / (float)n;
321:     Range = max - min;
322:     if (StartAtZero) {
323:       for (uint32_t i = 0; i < n; i++) {
324:         index = static_cast<uint32_t>(Data[i] / binRange);
325:         if (index == noBins) {
326:           index -= 1;
327:         }
328:         bins[index]++;
329:         sum_dev += pow((Data[i] - Mean), 2);
330:       }
331:     } else {
```

```
332:       for (uint32_t i = 0; i < n; i++) {
333:         index = static_cast<uint32_t>((Data[i] - min) / binRange);
334:         if (index == noBins) {
335:           index -= 1;
336:         }
337:         bins[index]++;
338:         sum_dev += pow((Data[i] - Mean), 2);
339:       }
340:     }
341:     Std = sqrt((float)(sum_dev / n));
342:     Calculated = true;
343:   }
344:
345:   /*!
346:    * \brief BasicCalculateFloat execute the basic float data calculations with a
347:    * mask
348:    * \param mask uchar mask type 0 don't calculate, 1 calculate
349:    */
350:   void BasicCalculateFloat(uchar *mask) {
351:     float sum_dev = 0.0;
352:     n = Rows * Cols;
353:     uint32_t nmask = 0;
354:     for (uint32_t i = 0; i < n; i++) {
355:       if (mask[i] != 0) {
356:         if (Data[i] > max) {
357:           max = Data[i];
358:         }
359:         if (Data[i] < min) {
360:           min = Data[i];
361:         }
362:         Sum += Data[i];
363:         nmask++;
364:       }
365:     }
366:     binRange = (max - min) / noBins;
367:     uint32_t index = 0;
368:     Mean = Sum / (float)nmask;
369:     Range = max - min;
370:     if (StartAtZero) {
371:       for (uint32_t i = 0; i < n; i++) {
372:         if (mask[i] != 0) {
373:           index = static_cast<uint32_t>(Data[i] / binRange);
374:           if (index == noBins) {
375:             index -= 1;
376:           }
377:           bins[index]++;
378:           sum_dev += pow((Data[i] - Mean), 2);
379:         }
380:       }
381:     } else {
382:       for (uint32_t i = 0; i < n; i++) {
383:         if (mask[i] != 0) {
384:           index = static_cast<uint32_t>((Data[i] - min) / binRange);
385:           if (index == noBins) {
386:             index -= 1;
387:           }
388:           bins[index]++;
389:           sum_dev += pow((Data[i] - Mean), 2);
390:         }
391:       }
392:     }
393:     Std = sqrt((float)(sum_dev / nmask));
394:     Calculated = true;
395:   }
396:
397:   /*!
398:    * \brief BasicCalculate execute the basic discrete data calculations
399:    */
400:   void BasicCalculate() {
401:     double sum_dev = 0.0;
402:     n = Rows * Cols;
403:     for (uint32_t i = 0; i < n; i++) {
404:       if (Data[i] > max) {
405:         max = Data[i];
406:       }
407:       if (Data[i] < min) {
408:         min = Data[i];
409:       }
410:       Sum += Data[i];
411:     }
412:     binRange = static_cast<T1>(ceil((max - min) / static_cast<float>(noBins)));
413:     Mean = Sum / (float)n;
414:     Range = max - min;
```

```
415:     uint32_t index;
416:     if (StartAtZero) {
417:       for_each(Data, Data + n, [&](T1 &d) {
418:         index = static_cast<uint32_t>(d / binRange);
419:         if (index == noBins) {
420:           index -= 1;
421:         }
422:         bins[index]++;
423:         sum_dev += pow((d - Mean), 2);
424:       });
425:     } else {
426:       for_each(Data, Data + n, [&](T1 &d) {
427:         index = static_cast<uint32_t>((d - min) / binRange);
428:         if (index == noBins) {
429:           index -= 1;
430:         }
431:         bins[index]++;
432:         sum_dev += pow((d - Mean), 2);
433:       });
434:     }
435:     Std = sqrt((float)(sum_dev / n));
436:     Calculated = true;
437:   }
438:
439:   /*!
440:    * \brief BasicCalculate execute the basic discrete data calculations with
441:    * mask
442:    * \param mask uchar mask type 0 don't calculate, 1 calculate
443:    */
444:   void BasicCalculate(uchar *mask) {
445:     double sum_dev = 0.0;
446:     n = Rows * Cols;
447:     uint32_t nmask = 0;
448:     uint32_t i = 0;
449:     for_each(Data, Data + n, [&](T1 &d) {
450:       if (mask[i++] != 0) {
451:         if (d > max) {
452:           max = d;
453:         }
454:         if (d < min) {
455:           min = d;
456:         }
457:         Sum += d;
458:         nmask++;
459:       }
460:     });
461:     binRange = static_cast<T1>(ceil((max - min) / static_cast<float>(noBins)));
462:     Mean = Sum / (float)nmask;
463:     Range = max - min;
464:     uint32_t index;
465:     if (StartAtZero) {
466:       i = 0;
467:       for_each(Data, Data + n, [&](T1 &d) {
468:         if (mask[i++] != 0) {
469:           index = static_cast<uint32_t>(d / binRange);
470:           if (index == noBins) {
471:             index -= 1;
472:           }
473:           bins[index]++;
474:           sum_dev += pow((d - Mean), 2);
475:         }
476:       });
477:     } else {
478:       i = 0;
479:       for_each(Data, Data + n, [&](T1 &d) {
480:         if (mask[i++] != 0) {
481:           index = static_cast<uint32_t>((d - min) / binRange);
482:           if (index == noBins) {
483:             index -= 1;
484:           }
485:           bins[index]++;
486:           sum_dev += pow((d - Mean), 2);
487:         }
488:       });
489:     }
490:     Std = sqrt((float)(sum_dev / nmask));
491:     Calculated = true;
492:   }
493:
494:   /*!
495:    * \brief BinCalculations excute the cacluations with the histogram
496:    * \param startC start counter
497:    * \param endC end counter
```

```
498:    */
499:   void BinCalculations(uint16_t startC, uint16_t endC __attribute__((unused))) {
500:     float sum_dev = 0.0;
501:     // Get the Sum
502:     uint32_t i = 0;
503:     for_each(begin(), end(), [&](uint32_t &b) { Sum += b * (startC + i++); });
504:
505:     // Get Mean
506:     Mean = Sum / (float)n;
507:
508:     // Get max
509:     for (int i = noBins - 1; i >= 0; i--) {
510:       if (bins[i] != 0) {
511:         max = i + startC;
512:         break;
513:       }
514:     }
515:
516:     // Get min
517:     for (uint32_t i = 0; i < noBins; i++) {
518:       if (bins[i] != 0) {
519:         min = i + startC;
520:         break;
521:       }
522:     }
523:
524:     // Get Max;
525:     Range = max - min;
526:
527:     // Calculate Standard Deviation
528:     i = 0;
529:     for_each(begin(), end(), [&](uint32_t &b) {
530:       sum_dev += b * pow(((i++ + startC) - Mean), 2);
531:     });
532:     Std = sqrt((float)(sum_dev / n));
533:     Calculated = true;
534:   }
535:
536: private:
537:   uint32_t n_end = 0; /**< data end counter used with mask*/
538:   friend class boost::serialization::access; /**< Serialization class*/
539:
540:   /*!
541:    * \brief serialize the object
542:    * \param ar argument
543:    * \param version
544:    */
545:   template <class Archive>
546:   void serialize(Archive &ar, const unsigned int version __attribute__((unused))) {
547:     ar &isDiscrete;
548:     ar &n;
549:     for (size_t dc = 0; dc < n; dc++) {
550:       ar &Data[dc];
551:     }
552:     ar &noBins;
553:     for (size_t dc = 0; dc < noBins; dc++) {
554:       ar &bins[dc];
555:     }
556:     ar &Calculated;
557:     ar &Mean;
558:     ar &Range;
559:     ar &min;
560:     ar &max;
561:     ar &Startbin;
562:     ar &EndBin;
563:     ar &binRange;
564:     ar &Std;
565:     ar &Sum;
566:     ar &Rows;
567:     ar &Cols;
568:     ar &StartAtZero;
569:   }
570: };
571: }
572:
573: typedef SoilMath::Stats<float, double, long double>
574:     floatStat_t; /**< floating Stat type*/
575: typedef SoilMath::Stats<uchar, uint32_t, uint64_t>
576:     ucharStat_t; /**< uchar Stat type*/
577: typedef SoilMath::Stats<uint16_t, uint32_t, uint64_t>
578:     uint16Stat_t; /**< uint16 Stat type*/
579: typedef SoilMath::Stats<uint32_t, uint32_t, uint64_t>
580:     uint32Stat_t; /**< uint32 Stat type*/
```

```
 1: /* Copyright (C) Jelle Spijker - All Rights Reserved
 2:  * Unauthorized copying of this file, via any medium is strictly prohibited
 3:  * and only allowed with the written consent of the author (Jelle Spijker)
 4:  * This software is proprietary and confidential
 5:  * Written by Jelle Spijker <spijker.jelle@gmail.com>, 2015
 6:  */
 7:
 8: /*! \brief Collection of the public SoilMath headers
 9:  *         Commonpractice is to include this header when you want to add
10:  * Soilmath routines
11:  */
12: #pragma once
13:
14: #include "Stats.h"
15: #include "Sort.h"
16: #include "FFT.h"
17: #include "NN.h"
18: #include "GA.h"
19: #include "CommonOperations.h"
20: #include "SoilMathTypes.h"
```

```cpp
/* Copyright (C) Jelle Spijker - All Rights Reserved
 * Unauthorized copying of this file, via any medium is strictly prohibited
 * and only allowed with the written consent of the author (Jelle Spijker)
 * This software is proprietary and confidential
 * Written by Jelle Spijker <spijker.jelle@gmail.com>, 2015
 */

#include "NN.h"

namespace SoilMath {
NN::NN() { beta = 0.666; }

NN::NN(uint32_t inputneurons, uint32_t hiddenneurons, uint32_t outputneurons) {
  // Set the number of neurons in the network
  inputNeurons = inputneurons;
  hiddenNeurons = hiddenneurons;
  outputNeurons = outputneurons;
  // Reserve the vector space
  iNeurons.reserve(inputNeurons + 1);  // input neurons + bias
  hNeurons.reserve(hiddenNeurons + 1); // hidden neurons + bias
  oNeurons.reserve(outputNeurons);     // output neurons

  beta = 0.666;
}

NN::~NN() {}

void NN::LoadState(string filename) {
  std::ifstream ifs(filename.c_str());
  boost::archive::xml_iarchive ia(ifs);
  ia >> boost::serialization::make_nvp("NeuralNet", *this);
}

void NN::SaveState(string filename) {
  std::ofstream ofs(filename.c_str());
  boost::archive::xml_oarchive oa(ofs);
  oa << boost::serialization::make_nvp("NeuralNet", *this);
}

Predict_t NN::PredictLearn(ComplexVect_t input, Weight_t inputweights,
                           Weight_t hiddenweights, uint32_t inputneurons,
                           uint32_t hiddenneurons, uint32_t outputneurons) {
  NN neural(inputneurons, hiddenneurons, outputneurons);
  neural.SetInputWeights(inputweights);
  neural.SetHiddenWeights(hiddenweights);
  return neural.Predict(input);
}

Predict_t NN::Predict(ComplexVect_t input) {
  if (input.size() != inputNeurons) {
    throw Exception::MathException("Size of input Neurons Exception!");
  }

  iNeurons.clear();
  hNeurons.clear();
  oNeurons.clear();

  // Set the bias in the input and hidden vector to 1 (real number)
  iNeurons.push_back(1.0f);
  hNeurons.push_back(1.0f);

  Predict_t retVal;
  uint32_t wCount = 0;

  // Init the network
  for (uint32_t i = 0; i < inputNeurons; i++) {
    iNeurons.push_back(static_cast<float>(abs(input[i])));
  }
  for (uint32_t i = 0; i < hiddenNeurons; i++) {
    hNeurons.push_back(0.0f);
  }
  for (uint32_t i = 0; i < outputNeurons; i++) {
    oNeurons.push_back(0.0f);
  }

  for (uint32_t i = 1; i < hNeurons.size(); i++) {
    wCount = i - 1;
    for (uint32_t j = 0; j < iNeurons.size(); j++) {
      hNeurons[i] += iNeurons[j] * iWeights[wCount];
      wCount += hNeurons.size() - 1;
    }
    hNeurons[i] = 1 / (1 + pow(2.71828f, (-hNeurons[i] * beta)));
  }
```

```
  84:
  85:   for (uint32_t i = 0; i < oNeurons.size(); i++) {
  86:     wCount = i;
  87:     for (uint32_t j = 0; j < hNeurons.size(); j++) {
  88:       oNeurons[i] += hNeurons[j] * hWeights[wCount];
  89:       wCount += oNeurons.size();
  90:     }
  91:     oNeurons[i] =
  92:         (2 / (1.0f + pow(2.71828f, (-oNeurons[i] * beta)))) -
  93:         1; // Shift plus scale so the learning function can be calculated
  94:   }
  95:
  96:   retVal.OutputNeurons = oNeurons;
  97:   return retVal;
  98: }
  99:
 100: void NN::Learn(InputLearnVector_t input, OutputLearnVector_t cat,
 101:                uint32_t noOfDescriptorsUsed __attribute__((unused))) {
 102:   SoilMath::GA optim(PredictLearn, inputNeurons, hiddenNeurons, outputNeurons);
 103:   ComplexVect_t inputTest;
 104:   std::vector<Weight_t> weights;
 105:   Weight_t weight(((inputNeurons + 1) * hiddenNeurons) +
 106:                       ((hiddenNeurons + 1) * outputNeurons),
 107:                   0);
 108:   // loop through each case and adjust the weights
 109:   optim.Evolve(input, weight, MinMaxWeight_t(-50, 50), cat, 1000, 50);
 110:
 111:   this->iWeights = Weight_t(
 112:       weight.begin(), weight.begin() + ((inputNeurons + 1) * hiddenNeurons));
 113:   this->hWeights = Weight_t(
 114:       weight.begin() + ((inputNeurons + 1) * hiddenNeurons), weight.end());
 115:   studied = true;
 116: }
 117: }
```

```cpp
/* Copyright (C) Jelle Spijker - All Rights Reserved
 * Unauthorized copying of this file, via any medium is strictly prohibited
 * and only allowed with the written consent of the author (Jelle Spijker)
 * This software is proprietary and confidential
 * Written by Jelle Spijker <spijker.jelle@gmail.com>, 2015
 */

#include "GA.h"

namespace SoilMath {
GA::GA() {}

GA::GA(NNfunctionType nnfunction, uint32_t inputneurons, uint32_t hiddenneurons,
       uint32_t outputneurons) {
  this->NNfuction = nnfunction;
  this->inputneurons = inputneurons;
  this->hiddenneurons = hiddenneurons;
  this->outputneurons = outputneurons;
}

GA::~GA() {}

void GA::Evolve(const ComplexVect_t &inputValues, Weight_t &weights,
                std::vector<Weight_t> &prevWeights, MinMaxWeight_t rangeweights,
                Predict_t goal, uint32_t maxGenerations, uint32_t popSize) {
  // Create the population
  uint32_t NOprevPopUsed =
      prevWeights.size() < popSize ? prevWeights.size() : popSize;
  Population_t pop = Genesis(weights, rangeweights, popSize - NOprevPopUsed);
  for (uint32_t i = 0; i < NOprevPopUsed; i++) {
    PopMember_t newMember;
    newMember.weights = prevWeights[i];
    for (uint32_t j = 0; j < newMember.weights.size(); j++) {
      newMember.weightsGen.push_back(
          ConvertToGenome<float>(newMember.weights[j], rangeweights));
    }
    pop.push_back(newMember);
  }
  float totalFitness = 0.0;
  for (uint32_t i = 0; i < maxGenerations; i++) {
    CrossOver(pop);
    Mutate(pop);
    totalFitness = 0.0;
    GrowToAdulthood(pop, inputValues, rangeweights, goal, totalFitness);
    if (SurvivalOfTheFittest(pop, totalFitness)) {
      break;
    }
  }

  weights = pop[0].weights;
}

void GA::Evolve(const InputLearnVector_t &inputValues, Weight_t &weights,
                MinMaxWeight_t rangeweights, OutputLearnVector_t &goal,
                uint32_t maxGenerations, uint32_t popSize) {
  // Create the population
  Population_t pop = Genesis(weights, rangeweights, popSize);
  float totalFitness = 0.0;
  for (uint32_t i = 0; i < maxGenerations; i++) {
    CrossOver(pop);
    Mutate(pop);
    totalFitness = 0.0;
    GrowToAdulthood(pop, inputValues, rangeweights, goal, totalFitness);
    if (SurvivalOfTheFittest(pop, totalFitness)) {
      break;
    }
  }
  weights = pop[0].weights;
}

Population_t GA::Genesis(const Weight_t &weights, MinMaxWeight_t rangeweights,
                         uint32_t popSize) {
  if (popSize < 1)
    return Population_t();

  Population_t pop;
  unsigned seed = std::chrono::system_clock::now().time_since_epoch().count();
  std::default_random_engine gen(seed);
  std::uniform_real_distribution<float> dis(rangeweights.first,
                                            rangeweights.second);

  for (uint32_t i = 0; i < popSize; i++) {
    PopMember_t I;
```

```
    for (uint32_t j = 0; j < weights.size(); j++) {
      I.weights.push_back(dis(gen));
      I.weightsGen.push_back(
          ConvertToGenome<float>(I.weights[j], rangeweights));
    }
    pop.push_back(I);
  }
  return pop;
}

void GA::CrossOver(Population_t &pop) {
  Population_t newPop; // create a new population
  PopMember_t newPopMembers[2];
  SplitGenome_t Split[2];

  for (uint32_t i = 0; i < pop.size(); i += 2) {

    for (uint32_t j = 0; j < pop[i].weights.size(); j++) {
      // Split A
      Split[0].first = bitset<CROSSOVER>(
          pop[i].weightsGen[j].to_string().substr(0, CROSSOVER));
      Split[0].second =
          bitset<GENE_MAX - CROSSOVER>(pop[i].weightsGen[j].to_string().substr(
              CROSSOVER, GENE_MAX - CROSSOVER));

      // Split B
      Split[1].first = bitset<CROSSOVER>(
          pop[i + 1].weightsGen[j].to_string().substr(0, CROSSOVER));
      Split[1].second = bitset<GENE_MAX - CROSSOVER>(
          pop[i + 1].weightsGen[j].to_string().substr(CROSSOVER,
                                                      GENE_MAX - CROSSOVER));

      // Mate A and B to AB and BA
      newPopMembers[0].weightsGen.push_back(
          Genome_t(Split[0].first.to_string() + Split[1].second.to_string()));
      newPopMembers[1].weightsGen.push_back(
          Genome_t(Split[1].first.to_string() + Split[0].second.to_string()));
    }
    newPop.push_back(newPopMembers[0]);
    newPop.push_back(newPopMembers[1]);
    newPopMembers[0].weightsGen.clear();
    newPopMembers[1].weightsGen.clear();
  }

  // Allow the top tiers population partners to mate again
  uint32_t halfN = pop.size() / 2;
  for (uint32_t i = 0; i < halfN; i++) {
    for (uint32_t j = 0; j < pop[i].weights.size(); j++) {
      Split[0].first = bitset<CROSSOVER>(
          pop[i].weightsGen[j].to_string().substr(0, CROSSOVER));
      Split[0].second =
          bitset<GENE_MAX - CROSSOVER>(pop[i].weightsGen[j].to_string().substr(
              CROSSOVER, GENE_MAX - CROSSOVER));

      Split[1].first = bitset<CROSSOVER>(
          pop[i + 2].weightsGen[j].to_string().substr(0, CROSSOVER));
      Split[1].second = bitset<GENE_MAX - CROSSOVER>(
          pop[i + 2].weightsGen[j].to_string().substr(CROSSOVER,
                                                      GENE_MAX - CROSSOVER));

      newPopMembers[0].weightsGen.push_back(
          Genome_t(Split[0].first.to_string() + Split[1].second.to_string()));
      newPopMembers[1].weightsGen.push_back(
          Genome_t(Split[1].first.to_string() + Split[0].second.to_string()));
    }
    newPop.push_back(newPopMembers[0]);
    newPop.push_back(newPopMembers[1]);
    newPopMembers[0].weightsGen.clear();
    newPopMembers[1].weightsGen.clear();
  }
  pop = newPop;
}

void GA::Mutate(Population_t &pop) {
  unsigned seed = std::chrono::system_clock::now().time_since_epoch().count();

  std::default_random_engine gen(seed);
  std::uniform_real_distribution<float> dis(0, 1);

  std::default_random_engine genGen(seed);
  std::uniform_int_distribution<int> disGen(0, (GENE_MAX - 1));

  for (uint32_t i = 0; i < pop.size(); i++) {
```

```
167:     for (uint32_t j = 0; j < pop[i].weightsGen.size(); j++) {
168:       if (dis(gen) < MUTATIONRATE) {
169:         pop[i].weightsGen[j][disGen(genGen)].flip();
170:       }
171:     }
172:   }
173: }
174:
175: void GA::GrowToAdulthood(Population_t &pop, const ComplexVect_t &inputValues,
176:                          MinMaxWeight_t rangeweights, Predict_t goal,
177:                          float &totalFitness) {
178:   for (uint32_t i = 0; i < pop.size(); i++) {
179:     for (uint32_t j = 0; j < pop[i].weightsGen.size(); j++) {
180:       pop[i].weights.push_back(
181:           ConvertToValue<float>(pop[i].weightsGen[j], rangeweights));
182:     }
183:     Weight_t iWeight(pop[i].weights.begin(),
184:                      pop[i].weights.begin() +
185:                          ((inputneurons + 1) * hiddenneurons));
186:     Weight_t hWeight(pop[i].weights.begin() +
187:                          ((inputneurons + 1) * hiddenneurons),
188:                      pop[i].weights.end());
189:     Predict_t results = NNfuction(inputValues, iWeight, hWeight, inputneurons,
190:                                   hiddenneurons, outputneurons);
191:     for (uint32_t j = 0; j < results.OutputNeurons.size(); j++) {
192:       pop[i].Fitness -= results.OutputNeurons[j] / goal.OutputNeurons[j];
193:     }
194:     pop[i].Fitness += results.OutputNeurons.size();
195:     totalFitness += pop[i].Fitness;
196:   }
197: }
198:
199: void GA::GrowToAdulthood(Population_t &pop,
200:                          const InputLearnVector_t &inputValues,
201:                          MinMaxWeight_t rangeweights, OutputLearnVector_t &goal,
202:                          float &totalFitness) {
203:   for (uint32_t i = 0; i < pop.size(); i++) {
204:     for (uint32_t j = 0; j < pop[i].weightsGen.size(); j++) {
205:       pop[i].weights.push_back(
206:           ConvertToValue<float>(pop[i].weightsGen[j], rangeweights));
207:     }
208:     Weight_t iWeight(pop[i].weights.begin(),
209:                      pop[i].weights.begin() +
210:                          ((inputneurons + 1) * hiddenneurons));
211:     Weight_t hWeight(pop[i].weights.begin() +
212:                          ((inputneurons + 1) * hiddenneurons),
213:                      pop[i].weights.end());
214:     for (uint32_t j = 0; j < inputValues.size(); j++) {
215:       Predict_t results = NNfuction(inputValues[j], iWeight, hWeight,
216:                                     inputneurons, hiddenneurons, outputneurons);
217:       for (uint32_t k = 0; k < results.OutputNeurons.size(); k++) {
218:         pop[i].Fitness -= results.OutputNeurons[k] / goal[j].OutputNeurons[k];
219:       }
220:       pop[i].Fitness += results.OutputNeurons.size();
221:     }
222:     pop[i].Fitness /= inputValues.size();
223:     totalFitness += pop[i].Fitness;
224:   }
225: }
226:
227: bool GA::SurvivalOfTheFittest(Population_t &pop, float &totalFitness) {
228:   bool retVal = false;
229:   uint32_t decimationCount = pop.size() / 2;
230:
231:   unsigned seed = std::chrono::system_clock::now().time_since_epoch().count();
232:   std::default_random_engine gen(seed);
233:
234:   std::sort(pop.begin(), pop.end(), PopMemberSort);
235:
236:   uint32_t i = ELITISME;
237:   while (pop.size() > decimationCount) {
238:     if (i >= pop.size()) {
239:       i = ELITISME;
240:     }
241:     std::uniform_real_distribution<float> dis(0, totalFitness);
242:     if (dis(gen) < pop[i].Fitness) {
243:       pop.erase(pop.begin() + i--);
244:       totalFitness -= pop[i].Fitness;
245:     }
246:     i++;
247:   }
248:
249:   if (pop[0].Fitness < END_ERROR) {
```

```
250:     retVal = true;
251:   }
252:   return retVal;
253: }
254: }
```

```
    1: /* Copyright (C) Jelle Spijker - All Rights Reserved
    2:  * Unauthorized copying of this file, via any medium is strictly prohibited
    3:  * and only allowed with the written consent of the author (Jelle Spijker)
    4:  * This software is proprietary and confidential
    5:  * Written by Jelle Spijker <spijker.jelle@gmail.com>, 2015
    6:  */
    7:
    8: #pragma once
    9:
   10: #include <vector>
   11: #include <complex>
   12: #include <cmath>
   13: #include <valarray>
   14: #include <array>
   15: #include <deque>
   16: #include <queue>
   17: #include <iterator>
   18: #include <algorithm>
   19: #include <stdint.h>
   20: #include <opencv2/core.hpp>
   21: #include "SoilMathTypes.h"
   22: #include "MathException.h"
   23:
   24: namespace SoilMath {
   25: /*!
   26:  * \brief Fast Fourier Transform class
   27:  * \details Use this class to transform a black and white blob presented as a
   28:  * cv::Mat with values 0 or 1 to a vector of complex values representing the Fourier
   29:  * Descriptors.
   30:  */
   31: class FFT {
   32: public:
   33:   /*!
   34:  * \brief Standard constructor
   35:  */
   36:   FFT();
   37:
   38:   /*!
   39:    * \brief Standard deconstructor
   40:    */
   41:   ~FFT();
   42:
   43:   /*!
   44:    * \brief Transforming the img to the frequency domain and returning the
   45:    * Fourier Descriptors
   46:    * \param img contour in the form of a cv::Mat type CV_8UC1. Which should
   47:    * consist of a continous contour. \f$ \{ img \in \mathbb{Z} | 0 \leq img \leq
   48:    * 1 \} \f$
   49:    * \return a vector with complex values, represing the contour in the
   50:    * frequency domain, expressed as Fourier Descriptors
   51:    */
   52:   ComplexVect_t GetDescriptors(const cv::Mat &img);
   53:
   54: private:
   55:   ComplexVect_t
   56:       fftDescriptors; /**< Vector with complex values which represent the
   57:                          descriptors*/
   58:   ComplexVect_t
   59:       complexcontour; /**< Vector with complex values which represent the
   60:                          contour*/
   61:   cv::Mat Img;        /**< Img which will be analysed*/
   62:
   63:   /*!
   64:    * \brief Contour2Complex a private function which translates a continous
   65:    * contour image
   66:    * to a vector of complex values. The contour is found using a depth first
   67:    * search with
   68:    * extension list. The alghorithm is based upon <a
   69:    * href="http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-034-artificial-intelligence-fall-2010/lecture-videos/lecture-4-search-depth-first-hill-climbing-beam/">MIT
   70:    * opencourseware
   71:    * 6-034-artificial-intelligence lecture 4</a>
   72:    * \param img contour in the form of a cv::Mat type CV_8UC1. Which should
   73:    * consist of a continous contour. \f$ \{ img \in \mathbb{Z} | 0 \leq img \leq
   74:    * 1 \} \f$
   75:    * \param centerCol centre of the contour X value
   76:    * \param centerRow centre of the contour Y value
   77:    * \return a vector with complex values, represing the contour as a function
   78:    */
   79:   ComplexVect_t Contour2Complex(const cv::Mat &img, float centerCol,
   80:                                 float centerRow);
   81:   /*!
   82:    * \brief Neighbors a private function returning the neighboring pixels which
```

```
 83:     * belong to a contour
 84:     * \param O uchar pointer to the data
 85:     * \param pixel current counter
 86:     * \param columns total number of columns
 87:     * \param rows total number of rows
 88:     * \return
 89:     */
 90:    iContour_t Neighbors(uchar *O, int pixel, uint32_t columns, uint32_t rows);
 91:
 92:    /*!
 93:     * \brief fft a private function calculating the Fast Fourier Transform
 94:     * let \f$ m \f$ be an integer and let \f$ N=2^m \f$ also
 95:     * \f$ CA=[x_0,\ldots,x_{N-1}] \f$ is an \f$ N \f$ dimensional complex vector
 96:     * let \f$ \omega=\exp({-2\pi i\over N}) \f$
 97:     * then \f$ c_k={\frac{1}{N}}\sum_{j=0}^{j=N-1}CA_j\omega^{jk} \f$
 98:     * \param CA a \f$ CA=[x_0,\ldots,x_{N-1}] \f$ is an \f$ N \f$ dimensional
 99:     * complex vector
100:     */
101:    void fft(ComplexArray_t &CA);
102:
103:    /*!
104:     * \brief ifft
105:     * \param CA
106:     */
107:    void ifft(ComplexArray_t &CA);
108: };
109: }
```

```
/* Copyright (C) Jelle Spijker - All Rights Reserved
 * Unauthorized copying of this file, via any medium is strictly prohibited
 * and only allowed with the written consent of the author (Jelle Spijker)
 * This software is proprietary and confidential
 * Written by Jelle Spijker <spijker.jelle@gmail.com>, 2015
 */

#pragma once
#define COMMONOPERATIONS_VERSION 1

#include <algorithm>
#include <stdint.h>

namespace SoilMath {
inline uint16_t MinNotZero(uint16_t a, uint16_t b) {
  if (a != 0 && b != 0) {
    return (a < b) ? a : b;
  } else {
    return (a > b) ? a : b;
  }
}

inline uint16_t Max(uint16_t a, uint16_t b) { return (a > b) ? a : b; }

inline uint16_t Max(uint16_t a, uint16_t b, uint16_t c, uint16_t d) {
  return (Max(a, b) > Max(c, d)) ? Max(a, b) : Max(c, d);
}

inline uint16_t Min(uint16_t a, uint16_t b) { return (a < b) ? a : b; }

inline uint16_t Min(uint16_t a, uint16_t b, uint16_t c, uint16_t d) {
  return (Min(a, b) > Min(c, d)) ? Min(a, b) : Min(c, d);
}

static inline double quick_pow10(int n) {
  static double pow10[19] = {1, 10, 100, 1000, 10000, 100000, 1000000, 10000000,
                             100000000, 1000000000, 10000000000, 100000000000,
                             1000000000000, 10000000000000, 100000000000000,
                             1000000000000000, 10000000000000000,
                             100000000000000000, 1000000000000000000};
  return pow10[n];
}


// Source:
// http://martin.ankerl.com/2012/01/25/optimized-approximative-pow-in-c-and-cpp/
static inline double fastPow(double a, double b) {
  union {
    double d;
    int x[2];
  } u = {a};
  u.x[1] = (int)(b * (u.x[1] - 1072632447) + 1072632447);
  u.x[0] = 0;
  return u.d;
}

static inline double quick_pow2(int n) {
  static double pow2[256] = {
      0,     1,     4,     9,     16,    25,    36,    49,    64,    81,
      100,   121,   144,   169,   196,   225,   256,   289,   324,   361,
      400,   441,   484,   529,   576,   625,   676,   729,   784,   841,
      900,   961,   1024,  1089,  1156,  1225,  1296,  1369,  1444,  1521,
      1600,  1681,  1764,  1849,  1936,  2025,  2116,  2209,  2304,  2401,
      2500,  2601,  2704,  2809,  2916,  3025,  3136,  3249,  3364,  3481,
      3600,  3721,  3844,  3969,  4096,  4225,  4356,  4489,  4624,  4761,
      4900,  5041,  5184,  5329,  5476,  5625,  5776,  5929,  6084,  6241,
      6400,  6561,  6724,  6889,  7056,  7225,  7396,  7569,  7744,  7921,
      8100,  8281,  8464,  8649,  8836,  9025,  9216,  9409,  9604,  9801,
      10000, 10201, 10404, 10609, 10816, 11025, 11236, 11449, 11664, 11881,
      12100, 12321, 12544, 12769, 12996, 13225, 13456, 13689, 13924, 14161,
      14400, 14641, 14884, 15129, 15376, 15625, 15876, 16129, 16384, 16641,
      16900, 17161, 17424, 17689, 17956, 18225, 18496, 18769, 19044, 19321,
      19600, 19881, 20164, 20449, 20736, 21025, 21316, 21609, 21904, 22201,
      22500, 22801, 23104, 23409, 23716, 24025, 24336, 24649, 24964, 25281,
      25600, 25921, 26244, 26569, 26896, 27225, 27556, 27889, 28224, 28561,
      28900, 29241, 29584, 29929, 30276, 30625, 30976, 31329, 31684, 32041,
      32400, 32761, 33124, 33489, 33856, 34225, 34596, 34969, 35344, 35721,
      36100, 36481, 36864, 37249, 37636, 38025, 38416, 38809, 39204, 39601,
      40000, 40401, 40804, 41209, 41616, 42025, 42436, 42849, 43264, 43681,
      44100, 44521, 44944, 45369, 45796, 46225, 46656, 47089, 47524, 47961,
      48400, 48841, 49284, 49729, 50176, 50625, 51076, 51529, 51984, 52441,
      52900, 53361, 53824, 54289, 54756, 55225, 55696, 56169, 56644, 57121,
      57600, 58081, 58564, 59049, 59536, 60025, 60516, 61009, 61504, 62001,
```

```
   84:       62500, 63001, 63504, 64009, 64516, 65025};
   85:   return pow2[(n >= 0) ? n : -n];
   86: }
   87:
   88: static inline long float2intRound(double d) {
   89:   d += 6755399441055744.0;
   90:   return reinterpret_cast<int &>(d);
   91: }
   92: }
```

```cpp
/* Copyright (C) Jelle Spijker - All Rights Reserved
 * Unauthorized copying of this file, via any medium is strictly prohibited
 * and only allowed with the written consent of the author (Jelle Spijker)
 * This software is proprietary and confidential
 * Written by Jelle Spijker <spijker.jelle@gmail.com>, 2015
 */

#include "FFT.h"

namespace SoilMath {
FFT::FFT() {}

FFT::~FFT() {}

ComplexVect_t FFT::GetDescriptors(const cv::Mat &img) {
  if (!fftDescriptors.empty()) {
    return fftDescriptors;
  }

  complexcontour = Contour2Complex(img, img.cols / 2, img.rows / 2);

  // Supplement the vector of complex numbers so that N = 2^m
  uint32_t N = complexcontour.size();
  double logN = log(static_cast<double>(N)) / log(2.0);
  if (floor(logN) != logN) {
    // Get the next power of 2
    double nextLogN = floor(logN + 1.0);
    N = static_cast<uint32_t>(pow(2, nextLogN));

    uint32_t i = complexcontour.size();
    // Append the vector with zeros
    while (i++ < N) {
      complexcontour.push_back(Complex_t(0.0, 0.0));
    }
  }

  ComplexArray_t ca(complexcontour.data(), complexcontour.size());
  fft(ca);
  fftDescriptors.assign(std::begin(ca), std::end(ca));
  return fftDescriptors;
}

iContour_t FFT::Neighbors(uchar *O, int pixel, uint32_t columns,
                          uint32_t rows) {
  long int LUT_nBore[8] = {-columns + 1, -columns, -columns - 1, -1,
                           columns - 1,  columns,  1 + columns,  1};
  iContour_t neighbors;
  uint32_t pEnd = rows * columns;
  uint32_t count = 0;
  for (uint32_t i = 0; i < 8; i++) {
    count = pixel + LUT_nBore[i];
    while (count >= pEnd && i < 8) {
      count = pixel + LUT_nBore[++i];
    }
    if (i >= 8) {
      break;
    }
    if (O[count] == 1)
      neighbors.push_back(count);
  }
  return neighbors;
}

ComplexVect_t FFT::Contour2Complex(const cv::Mat &img, float centerCol,
                                   float centerRow) {
  uchar *O = img.data;
  uint32_t pEnd = img.cols * img.rows;

  std::deque<std::deque<uint32_t>> sCont;
  std::deque<uint32_t> eList;

  // Initialize the queue
  for (uint32_t i = 0; i < pEnd; i++) {
    if (O[i] == 1) {
      std::deque<uint32_t> tmpQ;
      tmpQ.push_back(i);
      sCont.push_back(tmpQ);
      break;
    }
  }

  if (sCont.front().size() < 1) {
    throw Exception::MathException("No contour found in image!");
```

```
  84:   } // Exception handling
  85:
  86:   uint32_t prev = -1;
  87:
  88:   // Extend path on queue
  89:   for (uint32_t i = sCont.front().front(); i < pEnd;) {
  90:     iContour_t nBors =
  91:         Neighbors(O, i, img.cols, img.rows); // find neighboring pixels
  92:     std::deque<uint32_t> cQ = sCont.front(); // store first queue;
  93:     sCont.erase(sCont.begin());              // erase first queue from beginning
  94:     if (cQ.size() > 1) {
  95:       prev = cQ.size() - 2;
  96:     } else {
  97:       prev = 0;
  98:     }
  99:     // Loop through each neighbor
 100:     for (uint32_t j = 0; j < nBors.size(); j++) {
 101:       if (nBors[j] != cQ[prev]) // No backtracking
 102:       {
 103:         if (nBors[j] == cQ.front() && cQ.size() > 8) {
 104:           i = pEnd;
 105:         } // Back at first node
 106:         if (std::find(eList.begin(), eList.end(), nBors[j]) ==
 107:             eList.end()) // Check if this current route is extended elsewhere
 108:         {
 109:           std::deque<uint32_t> nQ = cQ;
 110:           nQ.push_back(nBors[j]); // Add the neighbor to the queue
 111:           sCont.push_front(nQ);   // add the sequence to the front of the queue
 112:         }
 113:       }
 114:     }
 115:     if (nBors.size() > 2) {
 116:       eList.push_back(i);
 117:     } // if there are multiple choices put current node in extension List
 118:     if (i != pEnd) {
 119:       i = sCont.front().back();
 120:     } // If it isn't the end set i to the last node of the first queue
 121:     if (sCont.size() == 0) {
 122:       throw Exception::MathException(
 123:           "No continuous contour found, or less then 8 pixels long!");
 124:     }
 125:   }
 126:
 127:   // convert the first queue to a complex normalized vector
 128:   Complex_t cPoint;
 129:   ComplexVect_t contour;
 130:   float col = 0.0;
 131:   // Normalize and convert the complex function
 132:   for_each(
 133:       sCont.front().begin(), sCont.front().end(),
 134:       [&img, &cPoint, &contour, &centerCol, &centerRow, &col](uint32_t &e) {
 135:         col = (float)((e % img.cols) - centerCol);
 136:         if (col == 0.0) {
 137:           cPoint.real(1.0);
 138:         } else {
 139:           cPoint.real((float)(col / centerCol));
 140:         }
 141:         cPoint.imag((float)((floorf(e / img.cols) - centerRow) / centerRow));
 142:         contour.push_back(cPoint);
 143:       });
 144:
 145:   return contour;
 146: }
 147:
 148: void FFT::fft(ComplexArray_t &CA) {
 149:   const size_t N = CA.size();
 150:   if (N <= 1) {
 151:     return;
 152:   }
 153:
 154:   //!< Divide and conquor
 155:   ComplexArray_t even = CA[std::slice(0, N / 2, 2)];
 156:   ComplexArray_t odd = CA[std::slice(1, N / 2, 2)];
 157:
 158:   fft(even);
 159:   fft(odd);
 160:
 161:   for (size_t k = 0; k < N / 2; ++k) {
 162:     Complex_t ct = std::polar(1.0, -2 * M_PI * k / N) * odd[k];
 163:     CA[k] = even[k] + ct;
 164:     CA[k + N / 2] = even[k] - ct;
 165:   }
 166: }
```

```
167:
168: void FFT::ifft(ComplexArray_t &CA) {
169:   CA = CA.apply(std::conj);
170:   fft(CA);
171:   CA = CA.apply(std::conj);
172:   CA /= CA.size();
173: }
174: }
```

```
  1: /* Copyright (C) Jelle Spijker - All Rights Reserved
  2:  * Unauthorized copying of this file, via any medium is strictly prohibited
  3:  * and only allowed with the written consent of the author (Jelle Spijker)
  4:  * This software is proprietary and confidential
  5:  * Written by Jelle Spijker <spijker.jelle@gmail.com>, 2015
  6:  */
  7:
  8: //! Genetic Algorithmes used for optimization problems
  9: /*!
 10:   * Use this class for optimization problems. It's currently optimized for
 11:  * Neural Network optimzation
 12:   */
 13: #pragma once
 14:
 15: #include <bitset>
 16: #include <random>
 17: #include <string>
 18: #include <algorithm>
 19: #include <chrono>
 20:
 21: #include "NN.h"
 22: #include "SoilMathTypes.h"
 23: #include "MathException.h"
 24:
 25: namespace SoilMath {
 26:
 27: class GA {
 28: public:
 29:   /*!
 30:  * \brief GA Standard constructor
 31:  */
 32:   GA();
 33:
 34:   /*!
 35:    * \brief GA Construction with a Neural Network initializers
 36:    * \param nnfunction the Neural Network prediction function which results will
 37:    * be optimized
 38:    * \param inputneurons the number of input neurons in the Neural Network don't
 39:    * count the bias
 40:    * \param hiddenneurons the number of hidden neurons in the Neural Network
 41:    * don't count the bias
 42:    * \param outputneurons the number of output neurons in the Neural Network
 43:    */
 44:   GA(NNfunctionType nnfunction, uint32_t inputneurons, uint32_t hiddenneurons,
 45:      uint32_t outputneurons);
 46:
 47:   /*!
 48:    * \brief GA standard de constructor
 49:    */
 50:   ~GA();
 51:
 52:   /*!
 53:    * \brief Evolve Darwin would be proud!!! This function creates a population
 54:    * and itterates
 55:    * through the generation till the maximum number off itterations has been
 56:    * reached of the
 57:    * error is acceptable
 58:    * \param inputValues complex vector with a reference to the inputvalues
 59:    * \param weights reference to the vector of weights which will be optimized
 60:    * \param prevWeights pointer to the pevious weight results
 61:    * \param rangeweights pointer to the range of weights, currently it doesn't
 62:    * support indivudal ranges
 63:    * this is because of the crossing
 64:    * \param goal target value towards the Neural Network prediction function
 65:    * will be optimized
 66:    * \param maxGenerations maximum number of itterations default value is 200
 67:    * \param popSize maximum number of population, this should be an even number
 68:    */
 69:   void Evolve(const ComplexVect_t &inputValues, Weight_t &weights,
 70:               std::vector<Weight_t> &prevWeights, MinMaxWeight_t rangeweights,
 71:               Predict_t goal, uint32_t maxGenerations = 200,
 72:               uint32_t popSize = 30);
 73:
 74:   /*!
 75:    * \brief Evolve Darwin would be proud!!! This function creates a population
 76:    * and itterates
 77:    * through the generation till the maximum number off itterations has been
 78:    * reached of the
 79:    * error is acceptable
 80:    * \param inputValues complex vector with a reference to the inputvalues
 81:    * \param weights reference to the vector of weights which will be optimized
 82:    * \param rangeweights reference to the range of weights, currently it doesn't
 83:    * support indivudal ranges
```

```
   84:    * this is because of the crossing
   85:    * \param goal target value towards the Neural Network prediction function
   86:    * will be optimized
   87:    * \param maxGenerations maximum number of itterations default value is 200
   88:    * \param popSize maximum number of population, this should be an even number
   89:    */
   90:   void Evolve(const InputLearnVector_t &inputValues, Weight_t &weights,
   91:               MinMaxWeight_t rangeweights, OutputLearnVector_t &goal,
   92:               uint32_t maxGenerations = 200, uint32_t popSize = 30);
   93:
   94: private:
   95:   NNfunctionType NNfuction; /**< The Neural Net work function*/
   96:   uint32_t inputneurons;    /**< the total number of input neurons*/
   97:   uint32_t hiddenneurons;   /**< the total number of hidden neurons*/
   98:   uint32_t outputneurons;   /**< the total number of output neurons*/
   99:
  100:   /*!
  101:    * \brief Genesis private function which is the spark of live, using a random
  102:    * seed
  103:    * \param weights a reference to the used Weight_t vector
  104:    * \param rangeweights pointer to the range of weights, currently it doesn't
  105:    * support indivudal ranges
  106:    * \param popSize maximum number of population, this should be an even number
  107:    * \return
  108:    */
  109:   Population_t Genesis(const Weight_t &weights, MinMaxWeight_t rangeweights,
  110:                        uint32_t popSize);
  111:
  112:   /*!
  113:    * \brief CrossOver a private function where the partners mate with each other
  114:    * The values or PopMember_t are expressed as bits or ar cut at the point CROSSOVER
  115:    * the population members are paired with the nearest neighbor and new members are
  116:    * created pairing the Genome_t of each other at the CROSSOVER point. Afterwards all
  117:    * the top tiers partners are allowed to mate again.
  118:    * \param pop reference to the population
  119:    */
  120:   void CrossOver(Population_t &pop);
  121:
  122:   /*!
  123:    * \brief Mutate a private function where individual bits from the Genome_t are mutated
  124:    * at a random uniform distribution event defined by the MUTATIONRATE
  125:    * \param pop reference to the population
  126:    */
  127:   void Mutate(Population_t &pop);
  128:
  129:   /*!
  130:    * \brief GrowToAdulthood a private function where the new population members serve as the
  131:    * the input for the Neural Network prediction function. The results are weight against
  132:    * the goal and this weight determine the fitness of the population member
  133:    * \param pop reference to the population
  134:    * \param inputValues complex vector with a reference to the inputvalues
  135:    * \param rangeweights pointer to the range of weights, currently it doesn't
  136:    * support indivudal ranges
  137:    * \param goal a Predict_t type with the expected value
  138:    * \param totalFitness a reference to the total population fitness
  139:    */
  140:   void GrowToAdulthood(Population_t &pop, const ComplexVect_t &inputValues,
  141:                        MinMaxWeight_t rangeweights, Predict_t goal,
  142:                        float &totalFitness);
  143:
  144:   /*!
  145:    * \brief GrowToAdulthood a private function where the new population members serve as the
  146:    * the input for the Neural Network prediction function. The results are weight against
  147:    * the goal and this weight determine the fitness of the population member
  148:    * \param pop reference to the population
  149:    * \param inputValues a InputLearnVector_t with a reference to the inputvalues
  150:    * \param rangeweights pointer to the range of weights, currently it doesn't
  151:    * support indivudal ranges
  152:    * \param goal a Predict_t type with the expected value
  153:    * \param totalFitness a reference to the total population fitness
  154:    */
  155:   void GrowToAdulthood(Population_t &pop, const InputLearnVector_t &inputValues,
  156:                        MinMaxWeight_t rangeweights, OutputLearnVector_t &goal,
  157:                        float &totalFitness);
  158:
  159:   /*!
  160:    * \brief SurvivalOfTheFittest a private function where a battle to the death commences
  161:    * The fittest population members have the best chance of survival. Death is instigated
  162:    * with a random uniform distibution. The elite members don't partake in this desctruction
  163:    * The ELITISME rate indicate how many top tier members survive this catastrophic event.
  164:    * \param inputValues a InputLearnVector_t with a reference to the inputvalues
  165:    * \param totalFitness a reference to the total population fitness
  166:    * \return
```

```
167:    */
168:   bool SurvivalOfTheFittest(Population_t &pop, float &totalFitness);
169:
170:   /*!
171:    * \brief PopMemberSort a private function where the members are sorted according to
172:    * there fitness ranking
173:    * \param i left hand population member
174:    * \param j right hand population member
175:    * \return true if the left member is closser to the goal as the right member.
176:    */
177:   static bool PopMemberSort(PopMember_t i, PopMember_t j) {
178:     return (i.Fitness < j.Fitness);
179:   }
180:
181:   /*!
182:    * \brief Conversion of the value of type T to Genome_t
183:    * \details Usage: Use <tt>ConvertToGenome<Type>(type, range)</tt>
184:    * \param value The current value wich should be converted to a Genome_t
185:    * \param range the range in which the value should fall, this is to have a Genome_t
186:    * which utilizes the complete range 0000...n till 1111...n
187:    */
188:   template <typename T>
189:   inline Genome_t ConvertToGenome(T value, std::pair<T, T> range) {
190:     uint32_t intVal = static_cast<uint32_t>(
191:         (UINT32_MAX * (range.first + value)) / (range.second - range.first));
192:     Genome_t retVal(intVal);
193:     return retVal;
194:   }
195:
196:   /*!
197:    * \brief Conversion of the Genome to a value
198:    * \details Usage: use <tt>ConvertToValue<Type>(genome, range)
199:    * \param gen is the Genome which is to be converted
200:    * \param range is the range in which the value should fall
201:    */
202:   template <typename T>
203:   inline T ConvertToValue(Genome_t gen, std::pair<T, T> range) {
204:     T retVal =
205:         range.first +
206:         (((range.second - range.first) * static_cast<T>(gen.to_ulong())) /
207:          UINT32_MAX);
208:     return retVal;
209:   }
210: };
211: }
```

```
1: /* Copyright (C) Jelle Spijker - All Rights Reserved
2:  * Unauthorized copying of this file, via any medium is strictly prohibited
3:  * and only allowed with the written consent of the author (Jelle Spijker)
4:  * This software is proprietary and confidential
5:  * Written by Jelle Spijker <spijker.jelle@gmail.com>, 2015
6:  */
7:
8: #pragma once
9: #include <exception>
10: #include <string>
11:
12: using namespace std;
13:
14: namespace SoilMath {
15: namespace Exception {
16: class MathException : public std::exception {
17: public:
18:   MathException(string m = "Math Exception!") : msg(m){};
19:   ~MathException() _GLIBCXX_USE_NOEXCEPT{};
20:   const char *what() const _GLIBCXX_USE_NOEXCEPT { return msg.c_str(); };
21:
22: private:
23:   string msg;
24: };
25: }
26: }
```

```
/* Copyright (C) Jelle Spijker - All Rights Reserved
 * Unauthorized copying of this file, via any medium is strictly prohibited
 * and only allowed with the written consent of the author (Jelle Spijker)
 * This software is proprietary and confidential
 * Written by Jelle Spijker <spijker.jelle@gmail.com>, 2015
 */

#pragma once
#include <stdint.h>

namespace SoilMath {
/*!
 * \brief The Sort template class
 */
class Sort {
public:
  Sort() {}
  ~Sort() {}

  /*!
   * \brief QuickSort a static sort a Type T array with i values
   * \details Usage: QuickSort<type>(*type , i)
   * \param arr an array of Type T
   * \param i the number of elements
   */
  template <typename T> static void QuickSort(T *arr, int i) {
    if (i < 2)
      return;

    T p = arr[i / 2];
    T *l = arr;
    T *r = arr + i - 1;
    while (l <= r) {
      if (*l < p) {
        l++;
      } else if (*r > p) {
        r--;
      } else {
        T t = *l;
        *l = *r;
        *r = t;
        l++;
        r--;
      }
    }
    Sort::QuickSort<T>(arr, r - arr + 1);
    Sort::QuickSort<T>(l, arr + i - l);
  }

  /*!
   * \brief QuickSort a static sort a Type T array with i values where the key
   * are also changed accordingly
   * \details Usage: QuickSort<type>(*type *type , i)
   * \param arr an array of Type T
   * \param key an array of 0..i-1 representing the index
   * \param i the number of elements
   */
  template <typename T> static void QuickSort(T *arr, T *key, int i) {
    if (i < 2)
      return;

    T p = arr[i / 2];

    T *l = arr;
    T *r = arr + i - 1;

    T *lkey = key;
    T *rkey = key + i - 1;

    while (l <= r) {
      if (*l < p) {
        l++;
        lkey++;
      } else if (*r > p) {
        r--;
        rkey--;
      } else {
       if (*l != *r) {
          T t = *l;
          *l = *r;
          *r = t;

          T tkey = *lkey;
```

```
   84:           *lkey = *rkey;
   85:           *rkey = tkey;
   86:         }
   87:
   88:         l++;
   89:         r--;
   90:
   91:         lkey++;
   92:         rkey--;
   93:       }
   94:     }
   95:     Sort::QuickSort<T>(arr, key, r - arr + 1);
   96:     Sort::QuickSort<T>(l, lkey, arr + i - l);
   97:   }
   98: };
   99: }
```

```
1: /* Copyright (C) Jelle Spijker - All Rights Reserved
2:  * Unauthorized copying of this file, via any medium is strictly prohibited
3:  * and only allowed with the written consent of the author (Jelle Spijker)
4:  * This software is proprietary and confidential
5:  * Written by Jelle Spijker <spijker.jelle@gmail.com>, 2015
6:  */
7:
8: #pragma once
9:
10: #include <stdint.h>
11: #include <vector>
12: #include <string>
13: #include <fstream>
14:
15: #include <boost/archive/xml_iarchive.hpp>
16: #include <boost/archive/xml_oarchive.hpp>
17: #include <boost/serialization/vector.hpp>
18:
19: #include "GA.h"
20: #include "MathException.h"
21: #include "SoilMathTypes.h"
22: #include "FFT.h"
23:
24: namespace SoilMath {
25: /*!
26:  * \brief The Neural Network class
27:  * \details This class is used to make prediction on large data set. Using self
28:  * learning algoritmes
29:  */
30: class NN {
31: public:
32:   /*!
33:  * \brief NN constructor for the Neural Net
34:  * \param inputneurons number of input neurons
35:  * \param hiddenneurons number of hidden neurons
36:  * \param outputneurons number of output neurons
37:  */
38:   NN(uint32_t inputneurons, uint32_t hiddenneurons, uint32_t outputneurons);
39:
40:   /*!
41:    * \brief NN constructor for the Neural Net
42:    */
43:   NN();
44:
45:   /*!
46:    * \brief ~NN virtual deconstructor for the Neural Net
47:    */
48:   virtual ~NN();
49:
50:   /*!
51:    * \brief Predict The prediction function.
52:    * \details In this function the neural net is setup and the input which are
53:    * the complex values descriping the contour in the frequency domein serve as
54:    * input. The absolute value of these im. number because I'm not interrested
55:    * in the orrientation of the particle but more in the degree of variations.
56:    * \param input vector of complex input values, these're the Fourier
57:    * descriptors
58:    * \return a real valued vector of the output neurons
59:    */
60:   Predict_t Predict(ComplexVect_t input);
61:
62:   /*!
63:    * \brief PredictLearn a static function used in learning of the weights
64:    * \details It starts a new Neural Network object and passes all the
65:    * paramaters in to this newly created object. After this the predict function
66:    * is called and the value is returned. This work around was needed to pass
67:    * the neural network to the Genetic Algorithm class.
68:    * \param input a complex vector of input values
69:    * \param inputweights the input weights
70:    * \param hiddenweights the hidden weights
71:    * \param inputneurons the input neurons
72:    * \param hiddenneurons the hidden neurons
73:    * \param outputneurons the output neurons
74:    * \return
75:    */
76:   static Predict_t PredictLearn(ComplexVect_t input, Weight_t inputweights,
77:                                 Weight_t hiddenweights, uint32_t inputneurons,
78:                                 uint32_t hiddenneurons, uint32_t outputneurons);
79:
80:   /*!
81:    * \brief SetInputWeights a function to set the input weights
82:    * \param value the real valued vector with the values
83:    */
```

```
  84:   void SetInputWeights(Weight_t value) { iWeights = value; }
  85:
  86:   /*!
  87:    * \brief SetHiddenWeights a function to set the hidden weights
  88:    * \param value the real valued vector with the values
  89:    */
  90:   void SetHiddenWeights(Weight_t value) { hWeights = value; }
  91:
  92:   /*!
  93:    * \brief SetBeta a function to set the beta value
  94:    * \param value a floating value ussualy between 0.5 and 1.5
  95:    */
  96:   void SetBeta(float value) { beta = value; }
  97:
  98:   /*!
  99:    * \brief Learn the learning function
 100:    * \param input a vector of vectors with complex input values
 101:    * \param cat a vector of vectors with the know output values
 102:    * \param noOfDescriptorsUsed the total number of descriptos which should be
 103:    * used
 104:    */
 105:   void Learn(InputLearnVector_t input, OutputLearnVector_t cat,
 106:              uint32_t noOfDescriptorsUsed);
 107:
 108:   /*!
 109:    * \brief SaveState Serialize and save the values of the Neural Net to disk
 110:    * \details Save the Neural Net in XML valued text file to disk so that a
 111:    * object can
 112:    * be reconstructed on a latter stadia.
 113:    * \param filename a string indicating the file location and name
 114:    */
 115:   void SaveState(string filename);
 116:
 117:   /*!
 118:    * \brief LoadState Loads the previouse saved Neural Net from disk
 119:    * \param filename a string indicating the file location and name
 120:    */
 121:   void LoadState(string filename);
 122:
 123:   Weight_t iWeights; /**< a vector of real valued floating point input weights*/
 124:   Weight_t hWeights; /**< a vector of real valued floating point hidden weight*/
 125:
 126: private:
 127:   std::vector<float> iNeurons; /**< a vector of input values, the bias is
 128:                                   included, the bias is included and
 129:                                     is the first value*/
 130:   std::vector<float>
 131:       hNeurons; /**< a vector of hidden values, the bias is included and
 132:                    is the first value*/
 133:   std::vector<float> oNeurons; /**< a vector of output values*/
 134:
 135:   uint32_t hiddenNeurons; /**< number of hidden neurons minus bias*/
 136:   uint32_t inputNeurons;  /**< number of input neurons minus bias*/
 137:   uint32_t outputNeurons; /**< number of output neurons*/
 138:   float beta; /**< the beta value, this indicates the steepness of the sigmoid
 139:                  function*/
 140:
 141:   bool studied =
 142:       false; /**< a value indicating if the weights are a results of a
 143:                 learning curve*/
 144:   friend class boost::serialization::access; /**< a private friend class so the
 145:                                                 serialization can access all
 146:                                                 the needed functions*/
 147:   /*!
 148:    * \brief serialization function
 149:    * \param ar the object
 150:    * \param version the version of the class
 151:    */
 152:   template <class Archive>
 153:   void serialize(Archive &ar, const unsigned int version __attribute__((unused))) {
 154:     ar &BOOST_SERIALIZATION_NVP(inputNeurons);
 155:     ar &BOOST_SERIALIZATION_NVP(hiddenNeurons);
 156:     ar &BOOST_SERIALIZATION_NVP(outputNeurons);
 157:     ar &BOOST_SERIALIZATION_NVP(iWeights);
 158:     ar &BOOST_SERIALIZATION_NVP(hWeights);
 159:     ar &BOOST_SERIALIZATION_NVP(beta);
 160:     ar &BOOST_SERIALIZATION_NVP(studied);
 161:   }
 162: };
 163: }
```