```cpp
#define BOOST_TEST_MODULE VSA_UNIT_TESTS

// Custom libraries
#include "../../../../src/VisionSoilAnalyzer/Vision/Vision.h"
#include "../../../../src/VisionSoilAnalyzer/Soil/VisionSoil.h"
#include "../../../../src/VisionSoilAnalyzer/SoilMath/SoilMath.h"
#include "FloatTestMatrix.h"
#include "TestMatrix.h"

#include <boost/test/unit_test.hpp>
#include <boost/test/results_reporter.hpp>
#include <iostream>
#include <fstream>

#include <opencv2/core/core.hpp>
#include <opencv2/highgui/highgui.hpp>

#include <string>
#include <boost/archive/xml_oarchive.hpp>
#include <boost/archive/xml_iarchive.hpp>
#include <boost/archive/binary_iarchive.hpp>
#include <boost/archive/binary_oarchive.hpp>
#include <boost/serialization/vector.hpp>

// Statistical analysis
#include <boost/math/distributions/students_t.hpp>
#include "StatisticalComparisonDefinition.h"

#include <math.h>
#include <cmath>
#include <random>
#include <sys/time.h>

using namespace cv;
using namespace std;

// Create the Report Redirector
struct LogToFile
{
    LogToFile()
    {
```

```
42          std::string logFileName(boost::unit_test::framework::master_test_suite().p_name);
43          logFileName.append(".log");
44          logFile.open(logFileName.c_str());
45          boost::unit_test::unit_test_log.set_stream(logFile);
46      }
47      ~LogToFile()
48      {
49          boost::unit_test::unit_test_log.test_finish();
50          logFile.close();
51          boost::unit_test::unit_test_log.set_stream(std::cout);
52      }
53      std::ofstream logFile;
54  };
55  BOOST_GLOBAL_FIXTURE(LogToFile);
56
57  struct M {
58      M()
59      {
60          BOOST_TEST_MESSAGE("setup fixture");
61          src = imread("../ComparisionPictures/SoilSampleRGB.ppm");
62      }
63      ~M()         { BOOST_TEST_MESSAGE("teardown fixture"); }
64
65      Mat src;
66      Mat dst;
67      Mat comp;
68  };
69
70  //Compare the sample using the Welch's Test (source: http://www.boost.org/doc/libs/1_57_0/libs/math/doc/html/math_toolkit/stat_tut/
      weg/st_eg/two_sample_students_t.html)
71  template <typename T1, typename T2, typename T3>
72  bool WelchTest(SoilMath::Stats<T1, T2, T3> &statComp, SoilMath::Stats<T1, T2, T3> &statDst)
73  {
74      double alpha = 0.05;
75      // Degrees of freedom:
76      double v = statComp.Std * statComp.Std / statComp.n + statDst.Std * statDst.Std / statDst.n;
77      v *= v;
78      double t1 = statComp.Std * statComp.Std / statComp.n;
79      t1 *= t1;
80      t1 /= (statComp.n - 1);
81      double t2 = statDst.Std * statDst.Std / statDst.n;
```

```
82      t2 *= t2;
83      t2 /= (statDst.n - 1);
84      v /= (t1 + t2);
85      // t-statistic:
86      double t_stat = (statComp.Mean - statDst.Mean) / sqrt(statComp.Std * statComp.Std / statComp.n + statDst.Std * statDst.Std /
          statDst.n);
87      //
88      // Define our distribution, and get the probability:
89      //
90      boost::math::students_t dist(v);
91      double q = cdf(complement(dist, fabs(t_stat)));
92
93      bool rejected = false;
94      // Sample 1 Mean == Sample 2 Mean test the NULL hypothesis, the two means are the same
95      if (q < alpha / 2)
96          rejected = false;
97      else
98          rejected = true;
99      return rejected;
100 }
101
102
103 //---------------------------------------------------------------------------------------
104 BOOST_AUTO_TEST_SUITE(SoilMath_Test_Suit)
105
106 BOOST_AUTO_TEST_CASE(SoilMath_ucharStat_t)
107 {
108     ucharStat_t Test((uint8_t *)testMatrix, 200, 200);
109
110     BOOST_CHECK_EQUAL_COLLECTIONS(Test.bins, Test.bins + 255, histTestResult, histTestResult + 255);
111     BOOST_CHECK_CLOSE(Test.Mean, meanTestResult, 0.0001);
112     BOOST_CHECK_EQUAL(Test.n, nTestResult);
113     BOOST_CHECK_EQUAL(Test.Sum, sumTestResult);
114     BOOST_CHECK_EQUAL(Test.min, minTestResult);
115     BOOST_CHECK_EQUAL(Test.max, maxTestResult);
116     BOOST_CHECK_EQUAL(Test.Range, rangeTestResult);
117     BOOST_CHECK_CLOSE(Test.Std, stdTestResult, 0.01);
118 }
119
120 BOOST_AUTO_TEST_CASE(SoilMath_floatStat_t)
121 {
```

```
122     floatStat_t Test((float *)ftestMatrix, 50, 50);
123
124     BOOST_CHECK_EQUAL_COLLECTIONS(Test.bins, Test.bins + 255, fhistTestResult, fhistTestResult + 255);
125     BOOST_CHECK_CLOSE(Test.Mean, fmeanTestResult, 0.01);
126     BOOST_CHECK_EQUAL(Test.n, fnTestResult);
127     BOOST_CHECK_CLOSE(Test.Sum, fsumTestResult, 0.01);
128     BOOST_CHECK_CLOSE(Test.min, fminTestResult, 0.01);
129     BOOST_CHECK_CLOSE(Test.max, fmaxTestResult, 0.01);
130     BOOST_CHECK_CLOSE(Test.Range, frangeTestResult, 0.01);
131     BOOST_CHECK_CLOSE(Test.Std, fstdTestResult, 0.025);
132 }
133
134 BOOST_AUTO_TEST_CASE(SoilMath_FFT_GetDescriptors)
135 {
136     uchar data[] =
137     { 0, 0, 1, 1, 0, 0,
138     0, 1, 0, 0, 1, 0,
139     0, 1, 0, 0, 1, 0,
140     1, 0, 0, 0, 0, 1,
141     1, 0, 0, 0, 0, 1,
142     0, 1, 0, 0, 1, 0,
143     0, 1, 0, 0, 1, 0,
144     0, 0, 1, 1, 0, 0 };
145     cv::Mat src(8, 6, CV_8UC1, &data, 1);
146     SoilMath::FFT Test;
147     ComplexVect_t desc = Test.GetDescriptors(src);
148     Complex_t desc_exp[] = {
149         Complex_t(-1.6666667, -6),
150         Complex_t(2.02375780, -0.16742019),
151         Complex_t(-3.1094758, -6.13316500),
152         Complex_t(-1.8036530, 1.023864110),
153         Complex_t(-0.6666667, -1),
154         Complex_t(0.04350554, -2.55884126),
155         Complex_t(-3.0522848, -0.63807119),
156         Complex_t(-0.7780360, -0.31809187),
157         Complex_t(-0.3333333, 0),
158         Complex_t(-1.3856866, -1.43021101),
159         Complex_t(-1.2238576, 0.466498316),
160         Complex_t(-0.3295119, 1.459385072),
161         Complex_t(1.33333331, -1),
```

```
162         Complex_t(-0.3482434, 0.489805636),
163         Complex_t(0.71895134, 0.304737878),
164         Complex_t(5.24453433, -0.49849056)
165     };
166
167     BOOST_CHECK_EQUAL(desc.size(), 16);
168
169     for (uint32_t i = 0; i < 16; i++)
170     {
171         BOOST_CHECK_CLOSE(desc[i].real(), desc_exp[i].real(), 0.0001);
172         BOOST_CHECK_CLOSE(desc[i].imag(), desc_exp[i].imag(), 0.0001);
173     }
174 }
175
176 BOOST_AUTO_TEST_CASE(SoilMath_FFT_GetDescriptors_Non_Continues_Contour)
177 {
178     uchar data[] =
179     { 0, 0, 0, 1, 0, 0,
180     0, 1, 0, 0, 1, 0,
181     0, 1, 0, 0, 1, 0,
182     1, 0, 0, 0, 0, 1,
183     1, 0, 0, 0, 0, 1,
184     0, 1, 0, 0, 1, 0,
185     0, 1, 0, 0, 1, 0,
186     0, 0, 1, 1, 0, 0 };
187     cv::Mat src(8, 6, CV_8UC1, &data, 1);
188     SoilMath::FFT Test;
189     BOOST_CHECK_THROW(Test.GetDescriptors(src), SoilMath::Exception::MathException);
190 }
191
192 BOOST_AUTO_TEST_CASE(SoilMath_NN_Save_And_Load)
193 {
194     SoilMath::NN Test(3, 5, 2);
195
196     InputLearnVector_t inputVect;
197     OutputLearnVector_t outputVect;
198
199     //Population_t pop;
200     unsigned seed = std::chrono::system_clock::now().time_since_epoch().count();
201     std::default_random_engine gen(seed);
202     std::uniform_real_distribution<float> dis(0.0, 1.0);
203
```

```
204     float i1 = 0.0, i2 = 0.0, i3 = 0.0;
205     float o1 = 0.0, o2 = 0.0;
206
207     for (uint32_t i = 0; i < 200; i++)
208     {
209         if (dis(gen) > 0.5f) { i1 = 1.0; }
210         else { i1 = 0.0; }
211         if (dis(gen) > 0.5f) { i2 = 1.0; }
212         else { i2 = 0.0; }
213         if (dis(gen) > 0.5f) { i3 = 1.0; }
214         else { i3 = 0.0; }
215
216         if (i1 == 1.0 && i2 == 1.0 && i3 == 0.0)
217         {
218             o1 = 1.0;
219             o2 = -1.0;
220         }
221         else if (i1 == 0.0 && i2 == 0.0 && i3 == 1.0)
222         {
223             o1 = 1.0;
224             o2 = -1.0;
225         }
226         else
227         {
228             o1 = -1.0;
229             o2 = 1.0;
230         }
231
232         ComplexVect_t inputTemp;
233         inputTemp.push_back(Complex_t(i1, 0));
234         inputTemp.push_back(Complex_t(i2, 0));
235         inputTemp.push_back(Complex_t(i3, 0));
236         inputVect.push_back(inputTemp);
237
238         Predict_t outputTemp;
239         outputTemp.OutputNeurons.push_back(o1);
240         outputTemp.OutputNeurons.push_back(o2);
241         outputVect.push_back(outputTemp);
242     }
243
244     Test.Learn(inputVect, outputVect, 0);
```

```
245     Test.SaveState("NN.xml");
246
247     SoilMath::NN loadTest;
248     loadTest.LoadState("NN.xml");
249
250     std::vector<float> test_out = Test.Predict(inputVect[0]).OutputNeurons;
251     std::vector<float> loadtest_out = loadTest.Predict(inputVect[0]).OutputNeurons;
252
253     BOOST_REQUIRE_EQUAL_COLLECTIONS(Test.hWeights.begin(), Test.hWeights.end(), loadTest.hWeights.begin(), loadTest.hWeights.end());
254     BOOST_REQUIRE_EQUAL_COLLECTIONS(Test.iWeights.begin(), Test.iWeights.end(), loadTest.iWeights.begin(), loadTest.iWeights.end());
255 }
256
257 BOOST_AUTO_TEST_CASE(SoilMath_NN_Prediction_Accurancy)
258 {
259     SoilMath::NN Test;
260     Test.LoadState("NN.xml");
261
262
263     InputLearnVector_t inputVect;
264     OutputLearnVector_t outputVect;
265     OutputLearnVector_t outputPredictVect;
266
267     unsigned seed = std::chrono::system_clock::now().time_since_epoch().count();
268     std::default_random_engine gen(seed);
269     std::uniform_real_distribution<float> dis(0.0, 1.0);
270
271     float i1 = 0.0, i2 = 0.0, i3 = 0.0;
272     float o1 = 0.0, o2 = 0.0;
273
274     for (uint32_t i = 0; i < 10; i++)
275     {
276         if (dis(gen) > 0.5f) { i1 = 1.0; }
277         else { i1 = 0.0; }
278         if (dis(gen) > 0.5f) { i2 = 1.0; }
279         else { i2 = 0.0; }
280         if (dis(gen) > 0.5f) { i3 = 1.0; }
281         else { i3 = 0.0; }
282
283         if (i1 == 1.0 && i2 == 1.0 && i3 == 0.0)
284         {
285             o1 = 1.0;
286             o2 = -1.0:
```

```
287         }
288         else if (i1 == 0.0 && i2 == 0.0 && i3 == 1.0)
289         {
290             o1 = 1.0;
291             o2 = -1.0;
292         }
293         else
294         {
295             o1 = -1.0;
296             o2 = 1.0;
297         }
298
299         ComplexVect_t inputTemp;
300         inputTemp.push_back(Complex_t(i1, 0));
301         inputTemp.push_back(Complex_t(i2, 0));
302         inputTemp.push_back(Complex_t(i3, 0));
303         inputVect.push_back(inputTemp);
304
305         Predict_t outputTemp;
306         outputTemp.OutputNeurons.push_back(o1);
307         outputTemp.OutputNeurons.push_back(o2);
308         outputVect.push_back(outputTemp);
309
310         Predict_t outputPredictTemp;
311         outputPredictTemp.OutputNeurons = Test.Predict(inputTemp).OutputNeurons;
312
313         for (uint32_t j = 0; j < outputTemp.OutputNeurons.size(); j++)
314         {
315             BOOST_CHECK_CLOSE(outputPredictTemp.OutputNeurons[j], outputTemp.OutputNeurons[j], 5);
316         }
317     }
318 }
319
320 BOOST_AUTO_TEST_SUITE_END()
321
322 //-----------------------------------------------------------------------------------------
323 BOOST_AUTO_TEST_SUITE(Vision_Test_Suite)
324
325 BOOST_FIXTURE_TEST_CASE(Vision_Convert_RGB_To_Intensity, M)
326 {
327     // Convert the RGB picture to an intensity picture
328     Vision::Conversion Test;
```

```
329     Test.Convert(src, dst, Vision::Conversion::RGB, Vision::Conversion::Intensity);
330
331     // Read in the Matlab converted intensity picture converted with the Matlab command:
332     // Matlab_int=0.2126*RGB(:,:,1)+0.7152*RGB(:,:,2)+0.0722*RGB(:,:,3);
333     comp = imread("../ComparisionPictures/Matlab_int.ppm", 0);
334
335     // Calculate the statistics of the two images
336     ucharStat_t statDst(dst.data, dst.rows, dst.cols);
337     ucharStat_t statComp(comp.data, comp.rows, comp.cols);
338
339     // Simple comparison
340     BOOST_CHECK_CLOSE(statDst.Mean, statComp.Mean, 0.5);
341     BOOST_CHECK_CLOSE(statDst.Std, statComp.Std, 0.5);
342     BOOST_CHECK_CLOSE((double)statDst.Range, (double)statComp.Range, 0.5);
343     BOOST_CHECK_CLOSE((double)statDst.min, (double)statComp.min, 0.5);
344     BOOST_CHECK_CLOSE((double)statDst.max, (double)statComp.max, 0.5);
345     BOOST_CHECK_CLOSE((double)statDst.Sum, (double)statComp.Sum, 0.5);
346
347     // Welch test comparison of the means
348     bool rejected = WelchTest<uchar, uint32_t, uint64_t>(statComp, statDst);
349     BOOST_CHECK_EQUAL(rejected, true);
350 }
351
352 BOOST_FIXTURE_TEST_CASE(Vision_Convert_RGB_To_CIEXYZ, M)
353 {
354     // Convert the RGB to an CIElab
355     Vision::Conversion Test;
356     Test.Convert(src, dst, Vision::Conversion::RGB, Vision::Conversion::CIE_XYZ);
357     vector<Mat> LAB = Test.extractChannel(dst, 0);
358
359     floatStat_t statDstX((float *)LAB[0].data, src.rows, src.cols);
360     floatStat_t statCompX;
361     statCompX.Std = X_STD;
362     statCompX.n = N_MAT;
363     statCompX.Mean = X_MEAN;
364     statCompX.Range = X_RANGE;
365     statCompX.min = X_MIN;
366     statCompX.max = X_MAX;
367     statCompX.Sum = X_SUM;
368
369     // Simple comparison
```

```
370     BOOST_CHECK_CLOSE(statDstX.Mean, statCompX.Mean, 0.5);
371     BOOST_CHECK_CLOSE(statDstX.Std, statCompX.Std, 0.5);
372     BOOST_CHECK_CLOSE((double)statDstX.Range, (double)statCompX.Range, 0.5);
373     BOOST_CHECK_CLOSE((double)statDstX.min, (double)statCompX.min, 0.5);
374     BOOST_CHECK_CLOSE((double)statDstX.max, (double)statCompX.max, 0.5);
375     BOOST_CHECK_CLOSE((double)statDstX.Sum, (double)statCompX.Sum, 0.5);
376
377     //// Welch test comparison of the means
378     //bool rejected = WelchTest<float, double, long double>(statCompX, statDstX);
379     //BOOST_CHECK_EQUAL(rejected, false); // TODO: Find out why my null hypothese doesn't hold
380
381     floatStat_t statDstY((float *)LAB[1].data, src.rows, src.cols);
382     floatStat_t statCompY;
383     statCompY.Std = Y_STD;
384     statCompY.n = N_MAT;
385     statCompY.Mean = Y_MEAN;
386     statCompY.Range = Y_RANGE;
387     statCompY.min = Y_MIN;
388     statCompY.max = Y_MAX;
389     statCompY.Sum = Y_SUM;
390
391     // Simple comparison
392     BOOST_CHECK_CLOSE(statDstY.Mean, statCompY.Mean, 0.5);
393     BOOST_CHECK_CLOSE(statDstY.Std, statCompY.Std, 0.5);
394     BOOST_CHECK_CLOSE((double)statDstY.Range, (double)statCompY.Range, 0.5);
395     BOOST_CHECK_CLOSE((double)statDstY.min, (double)statCompY.min, 0.5);
396     BOOST_CHECK_CLOSE((double)statDstY.max, (double)statCompY.max, 0.5);
397     BOOST_CHECK_CLOSE((double)statDstY.Sum, (double)statCompY.Sum, 0.5);
398
399     //// Welch test comparison of the means
400     //rejected = WelchTest<float, double, long double>(statCompY, statDstY);
401     //BOOST_CHECK_EQUAL(rejected, false);
402
403     floatStat_t statDstZ((float *)LAB[2].data, src.rows, src.cols);
404     floatStat_t statCompZ;
405     statCompZ.Std = Z_STD;
406     statCompZ.n = N_MAT;
407     statCompZ.Mean = Z_MEAN;
408     statCompZ.Range = Z_RANGE;
409     statCompZ.min = Z_MIN;
```

```
410     statCompZ.max = Z_MAX;
411     statCompZ.Sum = Z_SUM;
412
413     // Simple comparison
414     BOOST_CHECK_CLOSE(statDstZ.Mean, statCompZ.Mean, 0.5);
415     BOOST_CHECK_CLOSE(statDstZ.Std, statCompZ.Std, 0.5);
416     BOOST_CHECK_CLOSE((double)statDstZ.Range, (double)statCompZ.Range, 0.5);
417     BOOST_CHECK_CLOSE((double)statDstZ.min, (double)statCompZ.min, 0.5);
418     BOOST_CHECK_CLOSE((double)statDstZ.max, (double)statCompZ.max, 0.5);
419     BOOST_CHECK_CLOSE((double)statDstZ.Sum, (double)statCompZ.Sum, 0.5);
420
421     //// Welch test comparison of the means
422     //rejected = WelchTest<float, double, long double>(statCompZ, statDstZ);
423     //BOOST_CHECK_EQUAL(rejected, false);
424
425
426 }
427
428 BOOST_FIXTURE_TEST_CASE(Vision_Convert_RGB_To_CIElab, M)
429 {
430     // Convert the RGB to an CIElab
431     Vision::Conversion Test;
432     Test.Convert(src, dst, Vision::Conversion::RGB, Vision::Conversion::CIE_lab);
433     vector<Mat> LAB = Test.extractChannel(dst, 0);
434     imwrite("LAB.tiff", dst);
435
436     floatStat_t statDstL((float *)LAB[0].data, src.rows, src.cols);
437     floatStat_t statCompL;
438     statCompL.Std = L_STD;
439     statCompL.n = N_MAT;
440     statCompL.Mean = L_MEAN;
441     statCompL.Range = L_RANGE;
442     statCompL.min = L_MIN;
443     statCompL.max = L_MAX;
444     statCompL.Sum = L_SUM;
445
446     // Simple comparison
447     BOOST_CHECK_CLOSE(statDstL.Mean, statCompL.Mean, 0.5);
448     BOOST_CHECK_CLOSE(statDstL.Std, statCompL.Std, 0.5);
449     BOOST_CHECK_CLOSE((double)statDstL.Range, (double)statCompL.Range, 0.5);
450     BOOST_CHECK_CLOSE((double)statDstL.min, (double)statCompL.min, 0.5);
```

```
451     BOOST_CHECK_CLOSE((double)statDstL.max, (double)statCompL.max, 0.5);
452     BOOST_CHECK_CLOSE((double)statDstL.Sum, (double)statCompL.Sum, 0.5);
453
454     // Welch test comparison of the means
455     bool rejected = WelchTest<float, double, long double>(statCompL, statDstL);
456     BOOST_CHECK_EQUAL(rejected, false);
457
458
459     // Since the CIELa*b* values are doubles and they cannot be easily exported from Matlab. Thus the st.dev, n and mean are
          calculated in Matlab CieLab mat
460     // file is found in the comparison folder
461
462
463 }
464
465
466 BOOST_AUTO_TEST_SUITE_END()
467
468 //-----------------------------------------------------------------------------------------
469 BOOST_AUTO_TEST_SUITE(SoilAnalyzer_Test_Suite)
470
471 BOOST_FIXTURE_TEST_CASE(Soil_Sample_Save_And_Load, M)
472 {
473
474     SoilAnalyzer::Sample Test(src);
475     Test.Analyse();
476     std::string filename = "SoilSample.vsa";
477     Test.Save(filename);
478
479     SoilAnalyzer::Sample TestLoad;
480     TestLoad.Load(filename);
481
482     BOOST_CHECK_EQUAL_COLLECTIONS(Test.RGB.datastart, Test.RGB.dataend, TestLoad.RGB.datastart, TestLoad.RGB.dataend);
483 }
484
485 BOOST_AUTO_TEST_SUITE_END()
486
487
```