```
 1 /* ================================================================================================================= *
 2  * EPA ToxCast Challenge / TopCoder
 3  * PredictLEL v1 [Competition Sensitive]
 4  * RigelFive - Hudson Ohio USA
 5  * 12 May 2014 - 14 May 2014
 6  * ================================================================================================================= *
 7  * version history:
 8  * v0: Utilize Mw, atoms, bonds, chains, groups and rings as inputs.
 9  * v1: Utilize Mw as the only input.
10  *
11  * ================================================================================================================= *
12  * Atomized Software Specification:
13  * Customer Requirements:
14  *  1. Predict the LEL value for untested compounds using previous tests of other chemicals.
15  *
16  * Functional Requirements:
17  *  1. Estimate the LEL value based on information provided from untested compounds.
18  *
19  * Interface Requirements:
20  *  1. Output a double[] with the negative log value of the LEL estimated.
21  *
22  * Performance Requirements:
23  *  1. No time or memory requirements as no computations are required in the submitted code
24  *
25  * Design Requirements:
26  *  1. Utilize C++.
27  *  2. Model a NN to calculate the LEL value using a pseudo Joback-like method where
28  *     the main input parameters are: Mw, atoms, bonds, chains, groups and rings.
29  *  3. Determine generalized extrinsic parameters can be estimated from specific intrinsic parameters.
30  *  4. Do not use external databases related to the EPA or other agencies to determine the LEL values.
31  *
32  * Quality Requirements:
33  *  1. Maximize overall score of the algorithm (objective: 1,000,000 points).
34  *
35  * Verification Requirements:
36  *  1. Output 1854 values in a double[] array.
37  *
38  * ================================================================================================================= *
39  */
40
41 #define _LELPredictor_ 0; // output the version number
42
43 #include <cstdlib>
44 #include <cassert>              // remove
45 #include <cmath>                // remove
46 #include <fstream>              // remove
47 #include <sstream>              // remove
```

```
47 #include <sstream>              // remove
48 #include <algorithm>            // remove
49 #include <cstdio>
50 #include <ctime>                // remove
51 #include <vector>
52 #include <iostream>
53 #include <boost/tokenizer.hpp>  // remove
54
55 using namespace std;
56
57 struct Chemical {
58     string chemical_gs_id;
59     string chemical_casrn;
60     string chemical_name;
61     bool lel_value_known;
62     double systemic_adjusted_negative_log_lel;
63     string chemical_short_gs_id;   // truncate the "DSSTox_GSID_" and keep the id number in a string
64     string chemical_cid;     // matches the id in the DSSTox_Chemical and ToxCast_Chemical data, found in the DSSTox_Chemical_List
65     string chemical_SMILES;
66     float chemical_Mw;
67 };
68 struct DSSTox_Chemical {
69      string DSSTox_GSID;
70      string DSSTox_CID;
71      string TS_ChemName;
72      string TS_ChemName_Synonyms;
73      string TS_CASRN;
74      string ChemNote;
75      string STRUCTURE_Shown;
76      string STRUCTURE_Formula;
77      string STRUCTURE_MW;
78      string STRUCTURE_ChemType;
79      string STRUCTURE_IUPAC;
80      string STRUCTURE_SMILES;
81      string STRUCTURE_SMILES_Desalt;
82 };
83 struct ToxCast_Chemical {
84     string dsstox_cid; //DSSTox_CID
85 };
86
87 struct Connection {
88     double weight;
89     double deltaWeight;
90 };
91
92 class Neuron;
93 typedef vector<Neuron> Layer;
```

```
 94 class Neuron {
 95 public:
 96     Neuron(unsigned numOutputs, unsigned myIndex);
 97     void setOutputVal(double val) { m_outputVal = val; }
 98     double getOutputVal(void) const { return m_outputVal; }
 99     void feedForward(const Layer &prevLayer);
100     void calcOutputGradients(double targetVal);
101     void calcHiddenGradients(const Layer &nextLayer);
102     void updateInputWeights(Layer &prevLayer);
103
104 private:
105     static double eta;
106     static double alpha;
107     static double transferFunction(double x);
108     static double transferFunctionDerivative(double x);
109     static double randomWeight(void) { return rand() / double(RAND_MAX); }
110     double sumDOW(const Layer &nextLayer) const;
111     double m_outputVal;
112     vector<Connection> m_outputWeights;
113     unsigned m_myIndex;
114     double m_gradient;
115 };
116 double Neuron::eta = 0.15;
117 double Neuron::alpha = 0.50;
118 void Neuron::updateInputWeights(Layer &prevLayer) {
119     for (unsigned n = 0; n < prevLayer.size(); ++n) {
120         Neuron &neuron = prevLayer[n];
121         double oldDeltaWeight = neuron.m_outputWeights[m_myIndex].deltaWeight;
122         double newDeltaWeight =
123                 eta
124                 * neuron.getOutputVal()
125                 * m_gradient
126                 + alpha
127                 * oldDeltaWeight;
128         neuron.m_outputWeights[m_myIndex].deltaWeight = newDeltaWeight;
129         neuron.m_outputWeights[m_myIndex].weight += newDeltaWeight;
130     }
131 }
132 double Neuron::sumDOW(const Layer &nextLayer) const {
133     double sum = 0.0;
134
135     for (unsigned n = 0; n < nextLayer.size() - 1; ++n) {
136         sum += m_outputWeights[n].weight * nextLayer[n].m_gradient;
137     }
138     return sum;
139 }
140 void Neuron::calcHiddenGradients(const Layer &nextLayer) {
```

```
141     double dow = sumDOW(nextLayer);
142     m_gradient = dow * Neuron::transferFunctionDerivative(m_outputVal);
143 }
144 void Neuron::calcOutputGradients(double targetVal) {
145     double delta = targetVal - m_outputVal;
146     m_gradient = delta * Neuron::transferFunctionDerivative(m_outputVal);
147 }
148 double Neuron::transferFunction(double x) {
149     return tanh(x);
150 }
151 double Neuron::transferFunctionDerivative(double x) {
152     return 1.0 - x * x;
153 }
154 void Neuron::feedForward(const Layer &prevLayer) {
155     double sum = 0.0;
156 
157     for (unsigned n = 0; n < prevLayer.size(); ++n) {
158         sum += prevLayer[n].getOutputVal() *
159                 prevLayer[n].m_outputWeights[m_myIndex].weight;
160     }
161     m_outputVal = Neuron::transferFunction(sum);
162 }
163 Neuron::Neuron(unsigned numOutputs, unsigned myIndex) {
164     for (unsigned c = 0; c < numOutputs; ++c) {
165         m_outputWeights.push_back(Connection());
166         m_outputWeights.back().weight = randomWeight();
167     }
168     m_myIndex = myIndex;
169 }
170 
171 // ****************** class Net ******************
172 class Net
173 {
174 public:
175     Net(const vector<unsigned> &topology);
176     void feedForward(const vector<double> &inputVals);
177     void backProp(const vector<double> &targetVals);
178     void getResults(vector<double> &resultVals) const;
179     double getRecentAverageError(void) const { return m_recentAverageError; }
180 
181 private:
182     vector<Layer> m_layers; // m_layers[layerNum][neuronNum]
183     double m_error;
184     double m_recentAverageError;
185     static double m_recentAverageSmoothingFactor;
186 };
```

```
187
188 double Net::m_recentAverageSmoothingFactor = 100.0; // Number of training samples to average over
189
190 void Net::getResults(vector<double> &resultVals) const
191 {
192     resultVals.clear();
193
194     for (unsigned n = 0; n < m_layers.back().size() - 1; ++n) {
195         resultVals.push_back(m_layers.back()[n].getOutputVal());
196     }
197 }
198
199 void Net::backProp(const vector<double> &targetVals)
200 {
201     // Calculate overall net error (RMS of output neuron errors)
202
203     Layer &outputLayer = m_layers.back();
204     m_error = 0.0;
205
206     for (unsigned n = 0; n < outputLayer.size() - 1; ++n) {
207         double delta = targetVals[n] - outputLayer[n].getOutputVal();
208         m_error += delta * delta;
209     }
210     m_error /= outputLayer.size() - 1; // get average error squared
211     m_error = sqrt(m_error); // RMS
212
213     // Implement a recent average measurement
214
215     m_recentAverageError =
216             (m_recentAverageError * m_recentAverageSmoothingFactor + m_error)
217             / (m_recentAverageSmoothingFactor + 1.0);
218
219     // Calculate output layer gradients
220
221     for (unsigned n = 0; n < outputLayer.size() - 1; ++n) {
222         outputLayer[n].calcOutputGradients(targetVals[n]);
223     }
224
225     // Calculate hidden layer gradients
226
227     for (unsigned layerNum = m_layers.size() - 2; layerNum > 0; --layerNum) {
228         Layer &hiddenLayer = m_layers[layerNum];
229         Layer &nextLayer = m_layers[layerNum + 1];
230
231         for (unsigned n = 0; n < hiddenLayer.size(); ++n) {
232             hiddenLayer[n].calcHiddenGradients(nextLayer);
233         }
```

```
234     }
235
236     // For all layers from outputs to first hidden layer,
237     // update connection weights
238
239     for (unsigned layerNum = m_layers.size() - 1; layerNum > 0; --layerNum) {
240         Layer &layer = m_layers[layerNum];
241         Layer &prevLayer = m_layers[layerNum - 1];
242
243         for (unsigned n = 0; n < layer.size() - 1; ++n) {
244             layer[n].updateInputWeights(prevLayer);
245         }
246     }
247 }
248
249 void Net::feedForward(const vector<double> &inputVals)
250 {
251     assert(inputVals.size() == m_layers[0].size() - 1);
252
253     // Assign (latch) the input values into the input neurons
254     for (unsigned i = 0; i < inputVals.size(); ++i) {
255         m_layers[0][i].setOutputVal(inputVals[i]);
256     }
257
258     // forward propagate
259     for (unsigned layerNum = 1; layerNum < m_layers.size(); ++layerNum) {
260         Layer &prevLayer = m_layers[layerNum - 1];
261         for (unsigned n = 0; n < m_layers[layerNum].size() - 1; ++n) {
262             m_layers[layerNum][n].feedForward(prevLayer);
263         }
264     }
265 }
266
267 Net::Net(const vector<unsigned> &topology)
268 {
269     unsigned numLayers = topology.size();
270     for (unsigned layerNum = 0; layerNum < numLayers; ++layerNum) {
271         m_layers.push_back(Layer());
272         unsigned numOutputs = layerNum == topology.size() - 1 ? 0 : topology[layerNum + 1];
273
274         // We have a new layer, now fill it with neurons, and
275         // add a bias neuron in each layer.
276         for (unsigned neuronNum = 0; neuronNum <= topology[layerNum]; ++neuronNum) {
277             m_layers.back().push_back(Neuron(numOutputs, neuronNum));
278         }
279
```

```
280         // Force the bias node's output to 1.0 (it was the last neuron pushed in this layer):
281         m_layers.back().back().setOutputVal(1.0);
282     }
283 }
284
285 class ExampleChemicalData
286 {
287 public:
288     ExampleChemicalData(const string filename);
289     void getChemicalList(vector<string> &ChemicalList);
290     bool isEof(void) { return m_exampleChemicalDataFile.eof(); }
291 private:
292     ifstream m_exampleChemicalDataFile;
293 };
294 ExampleChemicalData::ExampleChemicalData(const string filename)
295 {
296     m_exampleChemicalDataFile.open(filename.c_str());
297 }
298 void ExampleChemicalData::getChemicalList(vector<string> &ChemicalList) {
299     string line;
300
301     getline(m_exampleChemicalDataFile, line);
302     ChemicalList.push_back(line);
303     return;
304 }
305
306 class DSSToxData
307 {
308 public:
309     DSSToxData(const string filename);
310     void getDSSToxList(vector<string> &DSSToxList);
311     bool isEof(void) { return m_exampleDSSToxDataFile.eof(); }
312 private:
313     ifstream m_exampleDSSToxDataFile;
314 };
315 DSSToxData::DSSToxData(const string filename)
316 {
317     m_exampleDSSToxDataFile.open(filename.c_str());
318 }
319 void DSSToxData::getDSSToxList(vector<string> &DSSToxList) {
320     string line;
321
322     getline(m_exampleDSSToxDataFile, line);
323     DSSToxList.push_back(line);
324     return;
325 }
326
```

```
327 class ToxCastData
328 {
329 public:
330     ToxCastData(const string filename);
331     void getToxCastList(vector<string> &ToxCastList);
332     bool isEof(void) { return m_exampleToxCastDataFile.eof(); }
333 private:
334     ifstream m_exampleToxCastDataFile;
335 };
336 ToxCastData::ToxCastData(const string filename)
337 {
338     m_exampleToxCastDataFile.open(filename.c_str());
339 }
340 void ToxCastData::getToxCastList(vector<string> &ToxCastList) {
341     string line;
342 
343     getline(m_exampleToxCastDataFile, line);
344     ToxCastList.push_back(line);
345     return;
346 }
347 
348 string chop_dsstox (string id_to_chop) {
349     // truncate the "DSSTox_GSID_" and keep the id number in a string
350 
351     stringstream ss(id_to_chop);
352     string blank1;
353     string blank2;
354     string id;
355 
356     getline(ss,blank1,'_') &&
357     getline(ss,blank2,'_') &&
358     getline(ss,id);
359 
360     return id;
361 }
362 
363 vector<string> tokenize(const string& line )
364 {
365     boost::escaped_list_separator<char> sep( '\\', ',', '"' ) ;
366     boost::tokenizer< boost::escaped_list_separator<char> > tokenizer( line, sep ) ;
367     return std::vector<std::string>( tokenizer.begin(), tokenizer.end() ) ;
368 }
369 
370 void output_c(Chemical c) {
371     cout << c.chemical_gs_id << " " <<
372             c.chemical_cid << " " <<
```

```
373             c.chemical_casrn << " " <<
374             c.chemical_name << " " <<
375             c.chemical_Mw << " " <<
376             c.chemical_SMILES  << " " <<
377             c.systemic_adjusted_negative_log_lel << endl;
378 }
379
380 void output_dc(DSSTox_Chemical dc) {
381     cout << dc.DSSTox_GSID << " " <<
382             dc.DSSTox_CID << " " <<
383             dc.TS_ChemName << " " <<
384             dc.TS_ChemName_Synonyms << " " <<
385             dc.TS_CASRN << " " <<
386             dc.ChemNote << " " <<
387             dc.STRUCTURE_Shown << " " <<
388             dc.STRUCTURE_Formula << " " <<
389             dc.STRUCTURE_MW << " " <<
390             dc.STRUCTURE_ChemType << " " <<
391             dc.STRUCTURE_IUPAC << " " <<
392             dc.STRUCTURE_SMILES << " " <<
393             dc.STRUCTURE_SMILES_Desalt << endl;
394 }
395
396 void output_tc(ToxCast_Chemical tc) {
397     cout << tc.dsstox_cid << " " <<
398             endl;
399 }
400
401 void showVectorVals(string label, vector<double> &v)
402 {
403     cout << label << " ";
404     for (unsigned i = 0; i < v.size(); ++i) {
405         cout << v[i] << " ";
406     }
407
408     cout << endl;
409 }
410
411 double sum(vector<double> v) {
412     double sum = 0.0;
413
414     int vSIZE = v.size();
415     for (int i = 0; i < vSIZE; i++) {
416         sum = sum + double(v.at(i));
417     }
418     return double(sum);
419 }
```

```
420 double average(vector<double> v) {
421     double sum = 0.0;
422 
423     int vSIZE = v.size();
424     for (int i = 0; i < vSIZE; i++) {
425         sum = sum + abs(v.at(i));
426     }
427     return double(sum/vSIZE);
428 }
429 double stddev2(vector<double> v) {
430     double s2;
431     double sum = 0.0;
432     double sum2 = 0.0;
433     double avg;
434 
435     int vSIZE = v.size();
436 
437     for (int i = 0; i < vSIZE; i++) {
438         sum = sum + abs(v.at(i));
439     }
440     avg = double(sum / vSIZE);
441 
442     for (int i = 0; i < vSIZE; i++) {
443         sum2 = sum2 + pow(((double) v.at(i) - avg),2.0);
444     }
445 
446     s2 = (1.0/((double) vSIZE-1.0)) * abs(sum2);
447 
448     return s2;
449 }
450 double min(vector<double> v) {
451     double min1 = 1e+09;
452     int vSIZE = v.size();
453 
454     for (int i = 0; i < vSIZE; i++) {
455         if (v.at(i) < min1) { min1 = v.at(i); };
456     }
457 
458     return double(min1);
459 }
460 double max(vector<double> v) {
461     double max1 = 0;
462 
463     int vSIZE = v.size();
464 
465     for (int i = 0; i < vSIZE; i++) {
```

```
466         if (v.at(i) > max1) { max1 = v.at(i); };
467     }
468
469     return double(max1);
470 }
471
472 double parabola(double a, double b, double c, double x) {
473     double y;
474
475     y = (a*pow((x-b),2.0)) + c;
476
477     return y;
478 }
479 double parabolic_filter(double x, double x_mean, double x_min, double x_max) {
480     double y, norm_x;
481     double a1, a2, b, c1, c2;
482
483     norm_x = x / x_max;
484
485     a1 = 1.0 / pow((x_mean-x_min),2.0);
486     a2 = 1.0 / pow((x_max-x_mean),2.0);
487     b = x_mean;
488     c1 = 0.0;
489     c2 = 0.0;
490
491     if (x < 0.0) {
492         y = -1.0;
493     } else if (x > x_max) {
494         y = 1.0;
495     } else {
496         if (x <= x_mean) {
497             y = -parabola(a1, b, c1, x);
498         }
499         else if (x > x_mean) {
500             y = parabola(a2, b, c2, x);
501         }
502     }
503     return y;
504 }
505 double parabolic_amplifier(double y, double x_mean, double x_min, double x_max) {
506
507     double x;
508     double a1, a2, b, c1, c2;
509
510     a1 = 1.0 / pow((x_mean-x_min),2.0);
511     a2 = 1.0 / pow((x_max-x_mean),2.0);
512     b = x_mean;
```

```
513     c1 = 0.0;
514     c2 = 0.0;
515 
516     if (y > 0) {
517         x = (1.0/(2.0*a2))*((2.0*a2*b)+(2.0*sqrt(a2*y-a2*c2)));
518     } else {
519         x = (1.0/(2.0*a1))*((2.0*a1*b)-(2.0*sqrt(-a1*y-a1*c1)));
520     }
521     return x;
522 }
523 
524 int main(int argc, char** argv) {
525 
526     string path = "/home/scott/Arctria/ARC-34 Yogi/v00/";
527     ExampleChemicalData exChemicalData("/home/scott/Arctria/ARC-34 Yogi/data/ToxCast_MM_Data/ToxRefDB_Challenge_Training.csv");
528     DSSToxData exDSSToxData("/home/scott/Arctria/ARC-34 Yogi/data/DSSTox/csv/TOX21S_v4a_8599_11Dec2013.csv");
529     ToxCastData exToxCastData("/home/scott/Arctria/ARC-34 Yogi/data/DSSTox/csv/toxprint_v2_vs_TOX21S_v4a_8599_03Dec2013.csv");
530 
531     // Initialize parameters
532     vector<double> the_big_answer;
533 
534     vector<string> tokenized_string_vector;
535 
536     vector<string> ChemicalList;
537     vector<string> DSS_ToxData_ChemicalList;
538     vector<string> ToxCast_ChemicalList;
539     vector<double> LEL_values;
540     vector<double> Mw_values;
541     vector<double> Predicted_LEL_values;
542 
543     vector<Chemical> Initial_Chemicals;
544     vector<Chemical> Training_Chemicals;
545     vector<Chemical> Predict_Chemicals;
546     vector<DSSTox_Chemical> DSS_Chemicals;
547     vector<ToxCast_Chemical> ToxCast_Chemicals;
548 
549     Chemical c;
550     DSSTox_Chemical dc;
551     ToxCast_Chemical tc;
552     Chemical pc;
553 
554     string in;
555     string blank_line;
556 
557     int val;
558     double d_val;
```

```
559
560     double Mw_min;
561     double Mw_mean;
562     double Mw_max;
563
564     double LEL_min;
565     double LEL_mean;
566     double LEL_max;
567
568     // LELPredict Program Timing Parameters for Training NN
569     bool startTraining = false;
570     bool finishTraining = false;
571     bool startCalculations = false;
572     bool finishCalculations = false;
573
574     // Program Timing Parameters
575     double training_start_time = 0.0;
576     double training_finish_time = 0.0;
577     double training_elapsed_time = 0.0;
578     double calculation_start_time = 0.0;
579     double calculation_finish_time = 0.0;
580     double calculation_elapsed_time = 0.0;
581
582     // Text analysis
583     bool quote_string = false;
584
585     for (int i = 1; i < argc; ++i) {
586         string arg = argv[i];
587         if ((arg == "-t") || (arg == "--training")) {
588             cout << "LELPredictor v00\nRigelFive - 12 May 2014" << endl;
589             cout << "================================================================================" << endl;
590
591             // Start the training
592             if (startTraining == false) {
593                 training_start_time = clock();
594                 startTraining = true;
595             }
596
597             // Read the .csv training file to build the Chemical List
598             while (!exChemicalData.isEof()) {
599                 exChemicalData.getChemicalList(ChemicalList);
600             }
601
602             // Parse the data using comma separated values from the Chemical List
603             for (int i = 1; i < ChemicalList.size()-1; i++) {
604                 in = ChemicalList.at(i);
605
```

```
606                 tokenized_string_vector.clear();
607                 boost::tokenizer<boost::escaped_list_separator<char> > tok(in);
608 
609                 for(boost::tokenizer<boost::escaped_list_separator<char> >::iterator beg=tok.begin(); beg!=tok.end();++beg){
610                     tokenized_string_vector.push_back(*beg);
611                 }
612 
613                 c.chemical_gs_id = tokenized_string_vector.at(0);
614                 c.chemical_casrn = tokenized_string_vector.at(1);
615                 c.chemical_name = tokenized_string_vector.at(2);
616                 c.chemical_short_gs_id = chop_dsstox(c.chemical_gs_id);
617                 c.chemical_cid = "-";
618                 c.chemical_SMILES = "-"; // temporary... changed to real value looked up from DSS_Chemicals
619                 c.chemical_Mw = 0.0; // temporary... changed to real value looked up from DSS_Chemicals
620 
621                 blank_line = tokenized_string_vector.at(3);
622 
623                 if (blank_line.find('?') != string::npos) {
624                     c.lel_value_known = false;
625                     c.systemic_adjusted_negative_log_lel = -1.0;
626                     Predict_Chemicals.push_back(c);
627                     Initial_Chemicals.push_back(c);
628 
629                 } else {
630                     c.lel_value_known = true;
631                     istringstream(blank_line) >> c.systemic_adjusted_negative_log_lel;
632                     LEL_values.push_back(c.systemic_adjusted_negative_log_lel);
633                     Training_Chemicals.push_back(c);
634                     Initial_Chemicals.push_back(c);
635                 }
636             }
637 
638             LEL_min = min(LEL_values);
639             LEL_mean = average(LEL_values);
640             LEL_max = max(LEL_values);
641 
642             cout << "Training Set Statistics:" << endl;
643             cout << "   Min LEL Value: " << min(LEL_values) << endl;
644             cout << "   Max LEL Value: " << LEL_max << endl;
645             cout << "   Average LEL Value: " << LEL_mean << endl;
646             cout << "   Sigma^2 of LEL Value: " << stddev2(LEL_values) << endl;
647             cout << "-------------------------------------------------------------------------------" << endl;
648 
649             // Build DSSTox_Chemical List
650             while (!exDSSToxData.isEof()) {
651                 exDSSToxData.getDSSToxList(DSS_ToxData_ChemicalList);
```

```
652             }
653
654             for (int i = 1; i < DSS_ToxData_ChemicalList.size()-1; i++) {
655                 in = DSS_ToxData_ChemicalList.at(i);
656                 tokenized_string_vector.clear();
657
658                 boost::escaped_list_separator<char> sep( '`', ',', '\"' ) ;         // apparently the character ` is never used!!!
659                 boost::tokenizer<boost::escaped_list_separator<char> > tok(in, sep);
660
661                 for(boost::tokenizer<boost::escaped_list_separator<char> >::iterator beg=tok.begin(); beg!=tok.end();++beg) {
662                     tokenized_string_vector.push_back(*beg);
663                 }
664
665                 dc.DSSTox_GSID = tokenized_string_vector.at(0);
666                 dc.DSSTox_CID = tokenized_string_vector.at(1);
667                 dc.TS_ChemName = tokenized_string_vector.at(2);
668                 dc.TS_ChemName_Synonyms = tokenized_string_vector.at(3);
669                 dc.TS_CASRN = tokenized_string_vector.at(4);
670                 dc.ChemNote = tokenized_string_vector.at(5);
671                 dc.STRUCTURE_Shown = tokenized_string_vector.at(6);
672                 dc.STRUCTURE_Formula = tokenized_string_vector.at(7);
673                 dc.STRUCTURE_MW = tokenized_string_vector.at(8);
674                 dc.STRUCTURE_ChemType = tokenized_string_vector.at(9);
675                 dc.STRUCTURE_IUPAC = tokenized_string_vector.at(10);
676                 dc.STRUCTURE_SMILES = tokenized_string_vector.at(11);
677                 dc.STRUCTURE_SMILES_Desalt = tokenized_string_vector.at(12);
678
679                 DSS_Chemicals.push_back(dc);
680
681                 istringstream(tokenized_string_vector.at(8)) >> d_val;
682                 Mw_values.push_back(d_val);
683             }
684
685             Mw_min = min(Mw_values);
686             Mw_mean = average(Mw_values);
687             Mw_max = max(Mw_values);
688
689             cout << "   Min Mw Value: " << min(Mw_values) << endl;
690             cout << "   Max Mw Value: " << Mw_max << endl;
691             cout << "   Average Mw Value: " << Mw_mean << endl;
692             cout << "   Sigma^2 of Mw Value: " << stddev2(Mw_values) << endl;
693             cout << "-------------------------------------------------------------------------------" << endl;
694
695             // Build ToxCast_Chemical List
696             while (!exToxCastData.isEof()) {
697                 exToxCastData.getToxCastList(ToxCast_ChemicalList);
698             }
```

```
699
700             for (int i = 1; i < ToxCast_ChemicalList.size()-1; i++) {
701                 in = ToxCast_ChemicalList.at(i);
702                 tokenized_string_vector.clear();
703
704                 boost::escaped_list_separator<char> sep( '\\', ',', '\"' ) ;
705                 boost::tokenizer<boost::escaped_list_separator<char> > tok(in, sep);
706
707                 for(boost::tokenizer<boost::escaped_list_separator<char> >::iterator beg=tok.begin(); beg!=tok.end();++beg) {
708                     tokenized_string_vector.push_back(*beg);
709                 }
710
711                 tc.dsstox_cid = tokenized_string_vector.at(0);
712
713                 ToxCast_Chemicals.push_back(tc);
714
715             }
716
717             // Output basic information about data in memory
718             cout << "   Total Chemicals Listed: " << ChemicalList.size()-2 << endl;     // reduced by 2 for the header line and footer line
719             cout << "   Size of Training Chemicals List: " << Training_Chemicals.size() << endl;
720             cout << "   Size of Prediction Chemicals List: " << Predict_Chemicals.size() << endl;
721             cout << "-------------------------------------------------------------------------------" << endl;
722             cout << "   Total DSS Tox Chemicals: " << DSS_Chemicals.size() << endl;
723             cout << "   Total ToxCast Chemicals: " << ToxCast_Chemicals.size() << endl;
724             cout << "-------------------------------------------------------------------------------" << endl;
725
726             // Verify that the training set is using a CAS Registry Number (pretty sure in the data guide)
727             int nocas = 0;
728             for (int i = 0; i < Training_Chemicals.size(); i++) {
729                 c = Training_Chemicals.at(i);
730                 in = c.chemical_casrn;
731                 if (in.find("NOCAS") != string::npos) {
732                     nocas++;
733                 }
734             }
735
736             cout << "   Number of non-CAS chemicals in the training set: " << nocas << endl;
737             cout << "-------------------------------------------------------------------------------" << endl;
738
739             cout << "Training Chemical SMILES Structures:" << endl;
740             cout << "-------------------------------------------------------------------------------" << endl;
741
742             int count_training_match_dsstox = 0;
743
744             for (int i = 0; i < Training_Chemicals.size(); i++) {
```

```
745                 c = Training_Chemicals.at(i);
746
747                 for (int k = 0; k < DSS_Chemicals.size(); k++) {
748                     dc = DSS_Chemicals.at(k);
749                     if (dc.DSSTox_GSID == c.chemical_short_gs_id) {
750                         c.chemical_cid = dc.DSSTox_CID;
751                         istringstream(dc.STRUCTURE_MW) >> c.chemical_Mw;
752                         c.chemical_SMILES = dc.STRUCTURE_SMILES;
753                         Training_Chemicals.at(i) = c;
754                         count_training_match_dsstox++;
755                     }
756                 }
757                 cout << c.chemical_SMILES << endl;
758             }
759
760             int count_predict_match_dsstox = 0;
761
762             cout << "Predict Chemical SMILES Structures:" << endl;
763             cout << "--------------------------------------------------------------------------------" << endl;
764
765             for (int i = 0; i < Predict_Chemicals.size(); i++) {
766                 c = Predict_Chemicals.at(i);
767
768                 for (int k = 0; k < DSS_Chemicals.size(); k++) {
769                     dc = DSS_Chemicals.at(k);
770                     if (dc.DSSTox_GSID == c.chemical_short_gs_id) {
771                         c.chemical_cid = dc.DSSTox_CID;
772                         istringstream(dc.STRUCTURE_MW) >> c.chemical_Mw;
773                         c.chemical_SMILES = dc.STRUCTURE_SMILES;
774                         Predict_Chemicals.at(i) = c;
775                         count_predict_match_dsstox++;
776                     }
777                 }
778
779                 cout << c.chemical_SMILES << endl;
780             }
781
782             vector<unsigned> topology;
783             topology.clear();
784             topology.push_back(1);  // 1 input:  Mw
785             topology.push_back(3); // 3 nodes in the hidden layer
786             topology.push_back(1);  // LEL value
787
788             cout << "Creating a <" <<
789                     topology.at(0) << " " <<
790                     topology.at(1) << " " <<
791                     topology.at(2) <<
```

```
                    topology.at(2) <<
                    "> neural network" << endl;

            Net myNet(topology);

            vector<double> inputVals, targetVals, resultVals;
            int trainingPass = 0;

            double NN_cycle_start = clock();
            double NN_cycle_stop = NN_cycle_start + (300*1000000.0);
            while (((myNet.getRecentAverageError() > 0.001) || (trainingPass < 5)) && (clock() < NN_cycle_stop)) {
                trainingPass++;
                for (int i = 0; i < Training_Chemicals.size(); i++) {
                    inputVals.clear();
                    targetVals.clear();

                    c = Training_Chemicals.at(i);

                    inputVals.push_back(parabolic_filter((double) c.chemical_Mw, Mw_mean, Mw_min, Mw_max));
                    targetVals.push_back(parabolic_filter((double) c.systemic_adjusted_negative_log_lel, LEL_mean, LEL_min, LEL_max));

                    myNet.feedForward(inputVals);
                    myNet.getResults(resultVals);

                    assert(targetVals.size() == topology.back());
                    myNet.backProp(targetVals);

                }
            }

            cout << "Neural Net Recent Average Error: " << myNet.getRecentAverageError() << endl;
            cout << "Total Training Passes:  " << trainingPass << endl;
            cout << "-------------------------------------------------------------------------------" << endl;

            // finish training calculations
            finishTraining = true;

            if (finishTraining == true) {
                training_finish_time = clock();
                training_elapsed_time = training_finish_time - training_start_time;
                cout << "training elapsed time: " << (training_elapsed_time / 1000000.0) << " sec" << endl;
                finishTraining = true;
                startCalculations = true;
            }

            // start calculations
            startCalculations = true;
```

```
838             if (startCalculations == true) {
839                 calculation_start_time = clock();
840                 startCalculations = false;
841             }
842 
843             // Calculate the LEL value using the trained NN
844             for (int i = 0; i < Predict_Chemicals.size(); i++) {
845                 inputVals.clear();
846                 c = Predict_Chemicals.at(i);
847 
848                 inputVals.push_back(parabolic_filter((double)c.chemical_Mw, Mw_mean, Mw_min, Mw_max));
849 
850                 myNet.feedForward(inputVals);
851                 myNet.getResults(resultVals);
852 
853                 c.systemic_adjusted_negative_log_lel = parabolic_amplifier(resultVals.at(0), LEL_mean, LEL_min, LEL_max);
854 
855                 Predicted_LEL_values.push_back(c.systemic_adjusted_negative_log_lel);
856                 Predict_Chemicals.at(i) = c;
857             }
858 
859             cout << "-------------------------------------------------------------------------------" << endl;
860 
861             LEL_min = min(Predicted_LEL_values);
862             LEL_mean = average(Predicted_LEL_values);
863             LEL_max = max(Predicted_LEL_values);
864 
865             cout << "Calculation / Prediction Set Statistics:" << endl;
866             cout << "   Min LEL Value: " << LEL_min << endl;
867             cout << "   Max LEL Value: " << LEL_max << endl;
868             cout << "   Average LEL Value: " << LEL_mean << endl;
869             cout << "   Sigma^2 of LEL Value: " << stddev2(Predicted_LEL_values) << endl;
870             cout << "-------------------------------------------------------------------------------" << endl;
871 
872             // calculate the answers based on training
873             finishCalculations = true;
874 
875             cout << "class LELPredictor { " << endl;
876             cout << "public:" << endl << "  vector<double> predictLEL(void);" << endl;
877             cout << "};" << endl;
878 
879             cout << "vector<double> LELPredictor::predictLEL(void) {" << endl;
880             cout << "   // Output the LEL values" << endl;
881             cout << "   vector<double> lel;" << endl << endl;
882 
883             for (int i = 0; i < Initial_Chemicals.size(); i++) {
884                 c = Initial_Chemicals.at(i);
```

```
885
886                 for (int j = 0; j < Predict_Chemicals.size(); j++) {
887                     pc = Predict_Chemicals.at(j);
888
889                     if(c.chemical_gs_id == pc.chemical_gs_id) {
890                         c.systemic_adjusted_negative_log_lel = pc.systemic_adjusted_negative_log_lel;
891                         Initial_Chemicals.at(i) = c;
892                     }
893                 }
894
895                 cout << "   lel.push_back(" << c.systemic_adjusted_negative_log_lel << ");" << endl;
896             }
897             cout << "   return lel;" << endl;
898             cout << "}" << endl << endl;
899
900             cout << "--------------------------------------------------------------------------------" << endl;
901
902             if (finishCalculations == true) {
903                 calculation_finish_time = clock();
904                 calculation_elapsed_time = calculation_finish_time - training_start_time;
905                 cout << "total elapsed time: " << (calculation_elapsed_time / 1000000.0) << " sec" << endl;
906                 cout << "================================================================================" << endl;
907                 finishCalculations = false;
908             }
909         }
910     }
911     return 0;
912 }
```