Analysis Of Nearest Neighbor Algorithm

by Luis A. López, 2005

The nearest neighbor algorithm is very sensitive to irrelevant features. One of the techniques used to mitigate this problem, is searching over all the possible features, and using the ones that get the best accuracy. This is a very costly operation, and therefore we need good search algorithms to reduce the search time and still get good classification accuracy. Here, I will analyze the performing time and accuracy of forward selection, backward elimination and my own search algorithm.

The algorithm of forward selection start off with small feature subsets, and progressively adds new features to these subsets, in order to increase the accuracy of the nearest neighbor. It performs worse than either the backward elimination or Luis' algorithm, on accuracy. However, it is much faster than the backward elimination algorithm. It requires about the same time to compute the best accuracy, as my own algorithm. For the small dataset, this algorithm got an accuracy of 87.7%, using the feature subset {1,3,7,8,10,12,13,17}. For the large dataset, it found that the feature subset {4,10,13,14,22,24} got the best accuracy of 89.6%.

Backward elimination is characterized by an upside-down pyramid; i.e. we start with all the possible features in one subset, and progressively eliminate features from the subset, if their elimination increases the nearest neighbor accuracy. It tends to be more accurate than the forward selection algorithm, because we're simply comparing subsets with many more elements. However, the increase in accuracy has the negative side effect of increasing the time required to run the algorithm. Thus, for the small dataset, it got an accuracy of 91.2%, using the feature subset {3,8,10,16,17}, whereas with the big dataset, it got an accuracy of 92.1%, using the feature subset {10,13,22,24}.

Luis' algorithm combines the speed of the forward selection algorithm and the accuracy of the backward elimination algorithm. I designed my personal algorithm with the idea that expanding the most accurate subset of features will improve the current accuracy. The resulting subset will contain all the features of the parent subset, plus a new feature, which hopefully will improve the nearest neighbor accuracy. Indeed, my algorithm tends to be more accurate than the forward selection algorithm, and has about the same accuracy as the backward selection algorithm. Using the small dataset, it got an accuracy of 91.2%, using the feature subset {3,8,10,16,17}. With the big dataset, the feature subset {10,13,22,24} was the best subset, with an accuracy of 92.1%.

Moreover, my algorithm does not expand feature subsets in the order of increasing number of subsets, like forward selection, or decreasing number of subsets, like backward elimination. Instead, all feature subsets are put in a priority queue, and selected for expansion based on their accuracy. Thus, the time to run the program is reduced if subsets, with few features, have greater accuracy; they will be selected for expansion over other more complex subsets with more features, and the calculation of the Euclidean distance will be simpler. On the other hand, if more accurate feature subsets tend to have more elements, my algorithm will behave like forward selection, gradually increasing the number of features in the subsets that it chooses to explore.

Three search algorithm were analyzed; each one with advantages and disadvantages. Forward selection is not as accurate as backward selection or my own algorithm. It is faster than backward selection, however. Backward selection and Luis' algorithm tend to have the same accuracy. My algorithm tends to be faster than both forward selection and backward selection algorithms.

Trace of the Nearest Neighbor Algorithm, with the Small Dataset, Using Forward Selection

(First Page)

Welcome to Luis Lopez's Feature Selection Algorithm.
Type the number of the algorithm you want to run.

1) Forward Selection
2) Backward Elimination
3) Luis' Special Algorithm

option: 1

This dataset has 17 features (not including the class attribution), with 600 instances.

Please wait while I normalize the data... Done!

Running nearest neighbor with all 17 features, using 'leaving-one-out' evaluation.

Beginning search.

Using feature(s) {1} accuracy is 45%
Using feature(s) {2} accuracy is 51.7%
Using feature(s) {3} accuracy is 72.8%
Using feature(s) {4} accuracy is 48.5%
Using feature(s) {5} accuracy is 46.2%
Using feature(s) {6} accuracy is 50.8%
Using feature(s) {7} accuracy is 48.3%
Using feature(s) {8} accuracy is 51.5%
Using feature(s) {9} accuracy is 47.8%
Using feature(s) {10} accuracy is 74.2%
Using feature(s) {11} accuracy is 47.2%
Using feature(s) {12} accuracy is 50.7%
Using feature(s) {13} accuracy is 49.3%
Using feature(s) {14} accuracy is 51%
Using feature(s) {15} accuracy is 50.3%
Using feature(s) {16} accuracy is 49.5%
Using feature(s) {17} accuracy is 51.8%

Feature set {10} was best, accuracy is 74.2%

…

(Last Page)

(Warning, accuracy has decreased! Continuing search in case of local maxima)
Feature set {10,3,7,12,1,17,8,13,14,16,15,4} was best, accuracy is 81.2%

Using feature(s) {10,3,7,12,1,17,8,13,14,16,15,4,9} accuracy is 78.8%

Using feature(s) {10,3,7,12,1,17,8,13,14,16,15,4,2} accuracy is 80.8%
Using feature(s) {10,3,7,12,1,17,8,13,14,16,15,4,11} accuracy is 80.2%
Using feature(s) {10,3,7,12,1,17,8,13,14,16,15,4,6} accuracy is 80.3%
Using feature(s) {10,3,7,12,1,17,8,13,14,16,15,4,5} accuracy is 79.7%

(Warning, accuracy has decreased! Continuing search in case of local maxima)
Feature set {10,3,7,12,1,17,8,13,14,16,15,4,2} was best, accuracy is 80.8%

Using feature(s) {10,3,7,12,1,17,8,13,14,16,15,4,2,9} accuracy is 78.2%
Using feature(s) {10,3,7,12,1,17,8,13,14,16,15,4,2,5} accuracy is 77.8%
Using feature(s) {10,3,7,12,1,17,8,13,14,16,15,4,2,11} accuracy is 77.7%
Using feature(s) {10,3,7,12,1,17,8,13,14,16,15,4,2,6} accuracy is 79.3%

(Warning, accuracy has decreased! Continuing search in case of local maxima)
Feature set {10,3,7,12,1,17,8,13,14,16,15,4,2,6} was best, accuracy is 79.3%

Using feature(s) {10,3,7,12,1,17,8,13,14,16,15,4,2,6,9} accuracy is 75.5%
Using feature(s) {10,3,7,12,1,17,8,13,14,16,15,4,2,6,5} accuracy is 78.8%
Using feature(s) {10,3,7,12,1,17,8,13,14,16,15,4,2,6,11} accuracy is 76.7%

(Warning, accuracy has decreased! Continuing search in case of local maxima)
Feature set {10,3,7,12,1,17,8,13,14,16,15,4,2,6,5} was best, accuracy is 78.8%

Using feature(s) {10,3,7,12,1,17,8,13,14,16,15,4,2,6,5,9} accuracy is 74.2%
Using feature(s) {10,3,7,12,1,17,8,13,14,16,15,4,2,6,5,11} accuracy is 73.5%

(Warning, accuracy has decreased! Continuing search in case of local maxima)
Feature set {10,3,7,12,1,17,8,13,14,16,15,4,2,6,5,9} was best, accuracy is 74.2%

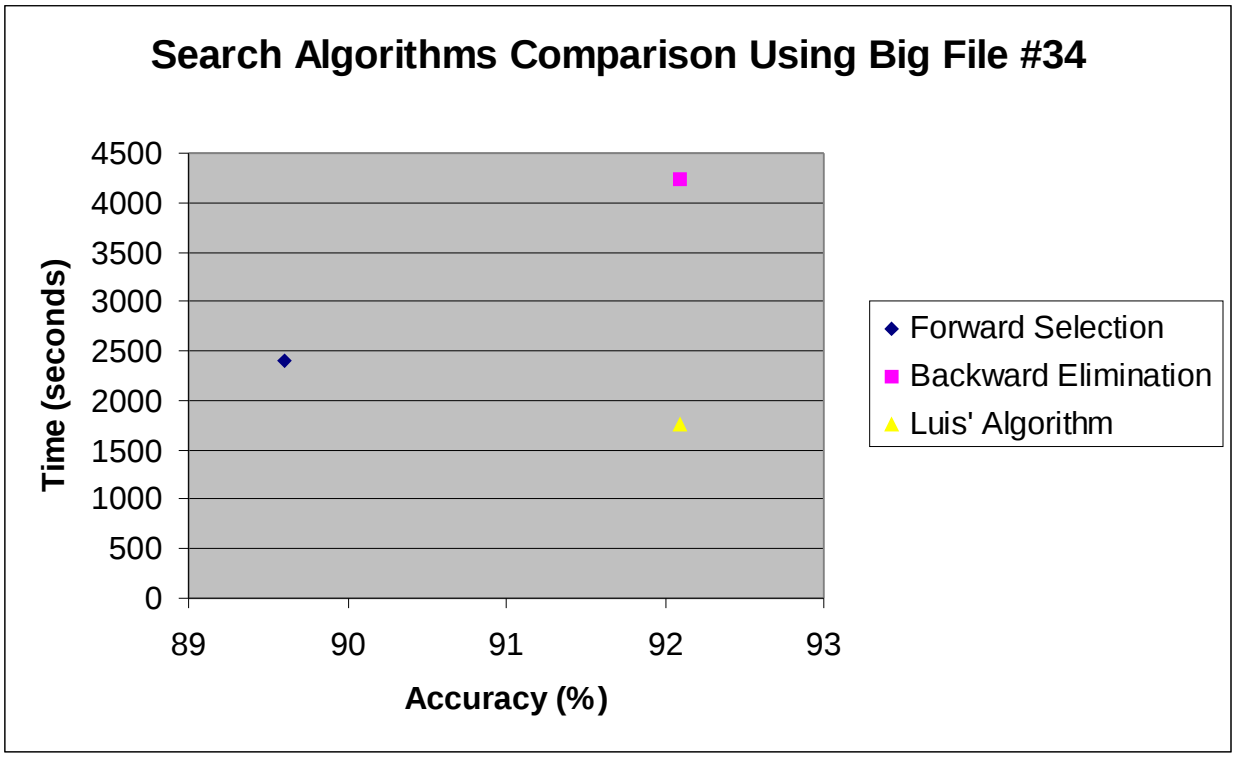
Using feature(s) {10,3,7,12,1,17,8,13,14,16,15,4,2,6,5,9,11} accuracy is 72.5%

(Warning, accuracy has decreased! Continuing search in case of local maxima)
Feature set {10,3,7,12,1,17,8,13,14,16,15,4,2,6,5,9,11} was best, accuracy is 72.5%

Finished search!! The best feature subset is {10,3,7,12,1,17,8,13}, which has an accuracy of 87.7%

Report

File			Forward Selection		Backward Elimination		Luis' Algorithm	
#	Number Features	Number Instance	User Time (sec)	Accuracy %	User Time (sec)	Accuracy %	User Time (sec)	Accuracy %
1	16	600	143.8	90.7	242.4	91.3	144	92.3
8	16	600	143.4	92	242.3	92	138.9	92
15	17	600	170.0	89.5	290.2	93.2	189	89.5
25	16	600	144.1	91	243.3	91.8	151.3	91
34	**17**	**600**	**184.6**	**87.7**	**316.7**	**91.2**	**177.9**	**91.2**
34	**29**	**1000**	**2406.1**	**89.6**	**4238.3**	**92.1**	**1758.9**	**92.1**
35	16	600	143.3	87.3	242.8	92.2	131.6	90.7
40	17	600	171	88.3	290.6	92.3	171.2	89.7
50	16	600	144	92	241.1	92	138.3	92
60	16	600	143.9	91.5	241.1	91.5	145.7	91.5



Code

Makefile

```
# CS 170: Project 2
# Luis A. Lopez
# SID: xxx-xx-xxxx
# Instructor: Dr. Eamonn Keogh


OBJS = main.o neighbor.o search.o
SRCS = main.cc neighbor.cc search.cc

nearestNeighbor: $(OBJS)
	g++ -g -o nearestNeighbor $(OBJS)

main.o: main.cc
	g++ -g -c -Wall main.cc

neighbor.o: neighbor.cc neighbor.h var.h
	g++ -g -c -Wall neighbor.cc

search.o: search.cc search.h var.h
	g++ -g -c -Wall search.cc

clean:
	rm -f nearestNeighbor *.o
```

main.cc

```
// CS 170: Project 2
// Luis A. Lopez
// SID: xxx-xx-xxxx
// Instructor: Dr. Eamonn Keogh


#include <iostream>
#include <string>
#include "neighbor.h"
#include "search.h"


using namespace std;


int main( int argc, char *argv[] )
{
```

```
  int opt=0;

  try {
    if (argc < 2) {
      throw string("format is 'nearestNeighbor <filename>'");
    }

    // Main menu
    cout << "\n\n";
    cout << "Welcome to Luis Lopez's Feature Selection Algorithm.\n";
    cout << "Type the number of the algorithm you want to run.\n";
    cout << "    1) Forward Selection\n";
    cout << "    2) Backward Elimination\n";
    cout << "    3) Luis' Special Algorithm\n";
    cout << "       option: ";
    cin >> opt;

    Neighbor ng;
    ng.ReadFile( argv[1] );
    if ( opt == 1 ) {   // Forward Selection
      ng.ForwardSelection();
    } else if ( opt == 2 ) {   // Backward Elimination
      ng.BackwardElimination();
    } else if ( opt == 3 ) {   // Luis' Special Algorithm
      Search s( &ng );
      s.GeneralSearch();
    } else
      throw string("invalid search option!");

  } catch( string e) {
    cout << "NearestNeighbor: " << e << endl;
  }

  return 0;
}
```

neighbor.h

```
// CS 170: Project 2
// Luis A. Lopez
// SID: xxx-xx-xxxx
// Instructor: Dr. Eamonn Keogh

#include <fstream>
#include <sstream>
#include "var.h"

#ifndef NEIGHBOR_H
#define NEIGHBOR_H

using namespace std;

class Neighbor {
  public:
    Neighbor();
    ~Neighbor();
    void ReadFile( char * );
    int GetNumFeatures();
    void ForwardSelection();
    void BackwardElimination();
    void LuisSpecialAlgorithm( Node, list<Node> &, set<int> &, double &, set<string> & );

  private:
    void NormalizeData();
    int EuclideanDistance( int [], int, int );
    string NumToString( double );

    vector< vector<double> > element;
    double **distMatrix;
    int rows, columns;
};

#endif
```

neighbor.cc

```
// CS 170: Project 2
// Luis A. Lopez
// SID: xxx-xx-xxxx
// Instructor: Dr. Eamonn Keogh

#include "neighbor.h"

using namespace std;

// Constructor: initializes data values
Neighbor::Neighbor() : rows( 0 ),
                       columns( 0 )
{
  cout.precision(3);
}


// Destructor
Neighbor::~Neighbor()
{
}



// Reads the file and stores it in a vector.  The first column describes the type
// of the element, and following columns are the features of this element.
void Neighbor::ReadFile( char *fileName )
{
  vector< double > newrow;
  ifstream file;
  string token="";
  char ch;
  int i=0, r=0, fileSize;

  // Open the file
  file.open(fileName, ifstream::in);
  // Get the size of the file
  file.seekg( 0, ios::end );
  fileSize = file.tellg();
  file.seekg( 0, ios::beg );
  // Check the file is open
  if (!file.is_open())
    throw string("the file could not be opened!");

  element.push_back(newrow);   // Create a new vector row
```

```
  while ( i < fileSize ) {    // Read the file
    token = "";

    ch = file.get();
    i++;
    while (ch != ' ' && ch != '\n' && i < fileSize) {    // New token found
      token += ch;
      ch = file.get();
      i++;
    }

    if (token != "")   // Insert token into the element vector
      element[r].push_back( atof(token.c_str()) );

    if (ch == '\n' && i < fileSize) {    // New data point - advance to the next vector row
      element.push_back(newrow);
      r++;
    }
  }

  rows = element.size();
  columns = element[0].size();

  cout << "\nThis dataset has " << columns-1 << " features (not including the class attribution),
with ";
  cout << rows << " instances.\n\n";

  NormalizeData();
}



// Z-normalizes the data
void Neighbor::NormalizeData()
{
  double dataMean;
  double stdDev;

  cout << "Please wait while I normalize the data... ";
  for (int j=1; j<columns; j++) {
    dataMean = 0.0;
    stdDev = 0.0;

    // Compute the mean
    for (int i=0; i<rows; i++) {
      dataMean += element[i][j];
```

```
    }
    dataMean = dataMean / rows;

    // Compute the standard deviation: s = sqrtRoot{Sigma[(x - mean)^2] / (n-1) }
    for (int i=0; i<rows; i++) {
      stdDev += (element[i][j] - dataMean) * (element[i][j] - dataMean);
    }
    stdDev = sqrt(stdDev / (rows-1));

    // Normalize the data for this column
    for (int i=0; i<rows; i++) {
      element[i][j] = (element[i][j] - dataMean) / stdDev;
    }
  }
  cout << "Done!\n\n";
}


// Compute the Euclidean distance between a new element to be classified
// and its neighbors.  For simplicity, the Euclidean distance is squared.
int Neighbor::EuclideanDistance( int featureList[], int size, int k )
{
  double distance=0.0, bestDistance=0.0;
  int closestNeighbor=0;
  bool isFirstNeighbor=true;

  for (int i=0; i < rows; i++) {   // Calculate the distance to each neighbor.
    distance = 0.0;

    if (i != k) {
      for (int j=0; j < size; j++) {   // Calculate the Euclidean destance with all the features.
        distance += (element[k][featureList[j]] - element[i][featureList[j]]) *
                    (element[k][featureList[j]] - element[i][featureList[j]]);
      }

      if (isFirstNeighbor) {   // Keep track of the closest neighbor.
        bestDistance = distance;
        closestNeighbor = i;
        isFirstNeighbor = false;
      } else if (distance < bestDistance) {
        bestDistance = distance;
        closestNeighbor = i;
      }
    }   // end if
  }   // end for
```

```
  if (element[k][0] == element[closestNeighbor][0])   // Return 1 if the new data point is
correctly classified.
    return 1;
  else
    return 0;
}


// Starts with an empty set and progressively selects new features,
// which improve the accuracy of the nearest neighbor algorithm.
void Neighbor::ForwardSelection()
{
  int featureList[columns-1], availFeatures[columns-1], bestFeature=0;
  double bestAccuracy=0.0, betterAccuracy=0.0, accuracy=0.0;
  vector<int> bestFeatureSubset;

  cout << "Running nearest neighbor with all " << columns-1 << " features, using 'leaving-one-
out' evaluation.\n\n";
  cout << "Beginning search.\n\n";

  for (int x=0; x < columns-1; x++)   // Store all the possible features in the availFeatures array
    availFeatures[x] = x+1;

  for (int i=1; i < columns; i++) {   // How many features to use
    betterAccuracy = 0.0;

    for (int j=0; j < columns-i; j++) {   // Traverse the available features
      featureList[i-1] = availFeatures[j];

      accuracy = 0.0;
      for (int k=0; k < rows; k++) {   // Use k-fold cross validation, w/ k=1 for each row of the
element matrix
        accuracy += EuclideanDistance( featureList, i, k );
      }
      accuracy = accuracy / rows * 100;

      cout << "        Using feature(s) {";   // Print accuracy results
      for (int x=0; x<i-1; x++)
        cout << featureList[x] << ",";
      cout << featureList[i-1] << "} accuracy is " << accuracy << "%" << endl;

      if (accuracy > betterAccuracy) {
        betterAccuracy = accuracy;
        bestFeature = j;
      }
    }   // end for
```

```
      featureList[i-1] = availFeatures[bestFeature];   // Expand the features with the best accuracy
only.
      availFeatures[bestFeature] = availFeatures[columns-1-i];   // Eliminate the selected feature
from ava. features.
      cout << endl;

      if (betterAccuracy > bestAccuracy) {   // Remember the best feature subset and its acuracy.
        bestAccuracy = betterAccuracy;
        bestFeatureSubset.clear();
        for (int x=0; x < i; x++)
          bestFeatureSubset.push_back(featureList[x]);
      } else
        cout << "(Warning, accuracy has decreased!  Continuing search in case of local maxima)"
<< endl;

      // Print the subset, which currently has the best accuracy.
      cout << "Feature set {";
      for (int x=0; x < i-1; x++)
        cout << featureList[x] << ",";
      cout << featureList[i-1] << "} was best, accuracy is  " << betterAccuracy << "%\n\n";
    }   // end for

    // Print the subset with the overall best accuracy.
    cout << "Finished search!!  The best feature subset is {";
    for (uint x=0; x < bestFeatureSubset.size()-1; x++)
      cout << bestFeatureSubset[x] << ",";
    cout << bestFeatureSubset[bestFeatureSubset.size()-1] << "}, which has an accuracy of " <<
bestAccuracy << "%\n";
}


// Start with all element in the set, and progressively eliminate elements,
// one at a time, to increase the accuracy of the nearest neighbor classifier.
void Neighbor::BackwardElimination()
{
    int featureList[columns-1], bestFeature=0, lastFeature=0, eliminatedFeature=0;
    double bestAccuracy=0.0, betterAccuracy=0.0, accuracy=0.0;
    vector<int> bestFeatureSubset;

    cout << "Running nearest neighbor with all " << columns-1 << " features, using 'leaving-one-
out' evaluation.\n\n";
    cout << "Beginning search.\n\n";

    for (int x=0; x < columns-1; x++)   // Store all the features in the featureList
      featureList[x] = x+1;
```

```
  for (int i=columns-1; i > 0; i--) {   // How many features to use
    betterAccuracy = 0.0;
    lastFeature = featureList[i];

    for (int j=i; j >= 0; j--) {   // Traverse the features
      // Eliminate another feature from the feature set, by swapping the values of
      // the last featureList value w/ that of the eliminated value.
      eliminatedFeature = featureList[j];
      featureList[j] = lastFeature;

      accuracy = 0.0;
      for (int k=0; k < rows; k++) {   // Use k-fold cross validation, w/ k=1 for each row of the
element matrix
        accuracy += EuclideanDistance( featureList, i, k );
      }
      accuracy = accuracy / rows * 100;

      cout << "        Using feature(s) {";   // Print accuracy results
      for (int x=0; x < i-1; x++)
        cout << featureList[x] << ",";
      cout << featureList[i-1] << "} accuracy is " << accuracy << "%" << endl;

      if (accuracy > betterAccuracy) {   // Remember the feature, whose elimination gives the
best accuracy.
        betterAccuracy = accuracy;
        bestFeature = j;
      }
      featureList[j] = eliminatedFeature; // Reset the featureList

      if (i == columns-1) {   // Don't eliminate any features, in the first iteration.
        j = -1;
      }
    }   // end for

    featureList[bestFeature] = lastFeature;   // Eliminate the feature, whose eliminate gave the
best accuracy.
    cout << endl;

    // Remember the better feature subset, among the subsets w/ the same number of elements,
and its accuracy.
    if (betterAccuracy > bestAccuracy) {
      bestAccuracy = betterAccuracy;
      bestFeatureSubset.clear();
      for (int x=0; x < i; x++)
        bestFeatureSubset.push_back(featureList[x]);
```

```
    } else
      cout << "(Warning, accuracy has decreased!  Continuing search in case of local maxima)"
<< endl;

    // Print the subset with the better accuracy.
    cout << "Feature set {";
    for (int x=0; x < i-1; x++)
      cout << featureList[x] << ",";
    cout << featureList[i-1] << "} was best, accuracy is  " << betterAccuracy << "%\n\n";
  }   // end for

  // Print the subset with the global best accuracy.
  cout << "Finished search!!  The best feature subset is {";
  for (uint x=0; x < bestFeatureSubset.size()-1; x++)
    cout << bestFeatureSubset[x] << ",";
  cout << bestFeatureSubset[bestFeatureSubset.size()-1] << "}, which has an accuracy of " <<
bestAccuracy << "%\n";
}


// Uses a priority queue to select the feature subset with the best accuracy.
// It then adds a new feature element to this subset, and enqueues the resulting subsets.
void Neighbor::LuisSpecialAlgorithm( Node node, list<Node> &successorNodes,
                    set<int> &bestFeatureSubset, double &bestAccuracy, set<string> &hashTable )
{
  int arraySize = node.featureList.size()+1;
  int featureList[arraySize], bestFeature=0, x;
  double betterAccuracy=0.0, accuracy=0.0;
  pair< set<int>::iterator, bool > isNewFeature;
  pair< set<string>::iterator, bool > isNewState;
  set< int >::iterator iter;

  // Copy the contents of the node set to a temp array.
  x=0;
  for (iter = node.featureList.begin(); iter != node.featureList.end(); iter++) {
    featureList[x] = *iter;
    x++;
  }

  betterAccuracy = 0.0;
  for (int i=1; i < columns; i++) {   // Traverse the list of features
    isNewFeature = node.featureList.insert(i);

    if (isNewFeature.second) {   // Put this feature subset in the hash table.
      node.state = "";
      for (iter = node.featureList.begin(); iter != node.featureList.end(); iter++)
```

```
        node.state += NumToString( *iter );
      isNewState = hashTable.insert(node.state);
    }

    if (isNewFeature.second && isNewState.second) {   // Verify this new feature is not already
in the set featureList.
      featureList[arraySize-1] = i;   // Store it in a temp array.
      accuracy = 0.0;

      for (int k=0; k < rows; k++) {   // Use k-fold cross validation, w/ k=1 for each row of the
element matrix
        accuracy += EuclideanDistance( featureList, arraySize, k );
      }
      accuracy = accuracy / rows * 100;   // Compute the accuracy
      node.value = accuracy;

      cout << "        Using feature(s) {";   // Print accuracy results
      for (int x=0; x < arraySize-1; x++)
        cout << featureList[x] << ",";
      cout << featureList[arraySize-1] << "} accuracy is " << accuracy << "%" << endl;

      if (accuracy > betterAccuracy) {   // Remember the better accuracy, so far.
        betterAccuracy = accuracy;
        bestFeature = i;
      }

      successorNodes.push_back(node);   // Store the node in the list of successor nodes
      node.featureList.erase(i);   // Reset the featureList set.
    } else if (isNewFeature.second && !isNewState.second)
      node.featureList.erase(i);
  }   // end for

  cout << endl;
  if (betterAccuracy > bestAccuracy) {   // Remember the best feature subset and its acuracy.
    bestAccuracy = betterAccuracy;
    bestFeatureSubset = node.featureList;
    bestFeatureSubset.insert( bestFeature );
  } else
    cout << "(Warning, accuracy has decreased!  Continuing search in case of local maxima)" <<
endl;
}


// Convert a double number to a string
string Neighbor::NumToString( double num )
{
```

```
  stringstream theString;
  theString << num;

  return theString.str();
}


// Return the total number of possible features
int Neighbor::GetNumFeatures()
{
  return (columns - 1);
}
```

var.h

```
// CS 170: Project 2
// Luis A. Lopez
// SID: xxx-xx-xxxx
// Instructor: Dr. Eamonn Keogh


#include <iostream>
#include <iomanip>
#include <string>
#include <vector>
#include <list>
#include <set>

#ifndef VAR_H
#define VAR_H

using namespace std;

struct Node {
  string state;   // State of the subset; for storing it in the hash table
  set<int> featureList;   // Subset of features to use
  double value;      // The value of the current state
};

#endif
```

search.h

```
// CS 170: Project 2
// Luis A. Lopez
// SID: xxx-xx-xxxx
// Instructor: Dr. Eamonn Keogh

#include "var.h"
#include "neighbor.h"

#ifndef SEARCH_H
#define SEARCH_H

using namespace std;


class Search {
  public:
    Search( Neighbor * );
    ~Search();
    void GeneralSearch();

  private:
    int MakeQueue( Node current, list<Node> &priorityQueue );
    Node RemoveFront( list<Node> &priorityQueue );
    void Expand( Node parentNode, list<Node> &successorNodes );
    void Enqueue( list<Node> &priorityQueue, list<Node> &successorNodes );

    Neighbor *neighbor;
    set< string > hashTable;
    set< int > bestFeatureSubset;
    double bestAccuracy;
    double prevBestAccuracy;
};

#endif
```

search.cc

```
// CS 170: Project 2
// Luis A. Lopez
// SID: xxx-xx-xxxx
// Instructor: Dr. Eamonn Keogh

#include "search.h"

using namespace std;

// Constructor: initializes data values
Search::Search( Neighbor *ng ) : bestAccuracy( 0.0 ),
                                prevBestAccuracy( 0.0 )
{
  neighbor = ng;
  bestFeatureSubset.clear();
  hashTable.clear();

  cout.precision(3);
}


// Destructor
Search::~Search()
{
}


// Algorithm used to search over a possible set of states.
void Search::GeneralSearch()
{
  Node node;
  list<Node> nodes, successorNodes;
  set< int >::iterator iter;
  int numFeatures, x;
  int size;

  numFeatures = neighbor->GetNumFeatures();

  cout << "Running nearest neighbor with all " << numFeatures << " features, using 'leaving-
one-out' evaluation.\n\n";
  cout << "Beginning search.\n\n";

  node.state = "";   // Initially, the first node in the queue does not contain any features.
  MakeQueue( node, nodes );
```

```
  int i=0;
  // Repeat until we have iterated a number of times (given by the # features) with no increase in
accuracy,
  // or the nodes list is empty.
  while ( (i < numFeatures) && !nodes.empty() ) {
    successorNodes.clear();

    node = RemoveFront( nodes );   // Compare the next node in the queue
    Expand( node, successorNodes );   // Expand the next node
    Enqueue( nodes, successorNodes );   // Put the successor nodes in the priority queue

    // Print out the best result of nearest neighbor, so far.
    node = nodes.front();
    cout << "Feature set {";
    x = 0;
    size = node.featureList.size();
    for (iter = node.featureList.begin(); iter != node.featureList.end(); iter++) {
      cout << *iter;
      if (x < size-1)
        cout << ",";
      x++;
    }
    cout << "} is currently the best, accuracy is  " << node.value << "%\n\n";

    if (node.value == 100 )   // Success: accuracy has reached 100%, exit the loop.
      i = numFeatures;

    i++;
  }   // end while.

  // Print out the overall best result for the nearest neighbor, using Luis Lopez' algorithm.
  cout << "Finished search!!  The best feature subset is {";
  x = 0;
  size = bestFeatureSubset.size();
  for (iter = bestFeatureSubset.begin(); iter != bestFeatureSubset.end(); iter++) {
    cout << *iter;
    if (x < size-1)
      cout << ",";
    x++;
  }
  cout << "}, which has an accuracy of " << bestAccuracy << "%\n";
}


// Insert the first node into the queue.
int Search::MakeQueue( Node current, list<Node> &nodeQueue )
```

```
{
  nodeQueue.push_back(current);

  return 0;
}


// Removes the first element from the queue, and returns it.
Node Search::RemoveFront( list<Node> &nodeQueue )
{
  Node front;

  front = nodeQueue.front();
  nodeQueue.pop_front();

  return front;
}


// Expand the next node
void Search::Expand( Node parentNode, list<Node> &successorNodes )
{
  neighbor->LuisSpecialAlgorithm( parentNode, successorNodes, bestFeatureSubset,
bestAccuracy, hashTable );
}

// Insert the list of successor nodes into the priority queue, according to their value.
// Greater values (i.e. more accuracy) receive more priority.
void Search::Enqueue( list<Node> &priorityQueue, list<Node> &successorNodes )
{
  list<Node>::iterator priorityIter=NULL;
  list<Node>::iterator successorIter=NULL;

  // Traverse the list of successor nodes
  for ( successorIter = successorNodes.begin(); successorIter != successorNodes.end();
successorIter++ ) {
    // Traverse the priority queue until we find the correct postion to insert the new node.
    for ( priorityIter = priorityQueue.begin();
        priorityIter != priorityQueue.end() && (*priorityIter).value >= (*successorIter).value;
        priorityIter++ ) {
      ;
    }
    priorityIter = priorityQueue.insert(priorityIter, (*successorIter));
  }
}
```