B.Comp Dissertation

Recognition of Metal Ion Binding Proteins

By
Edwin Jose Palathinkal

Department of Computer Science
School of Computing
National University of Singapore
2010/11

B.Comp Dissertation

Recognition of Metal Ion Binding Proteins

By
Edwin Jose Palathinkal

Department of Computer Science
School of Computing
National University of Singapore
2010/11

Project No: H114240
Advisor: Limsoon Wong
Deliverables:
- Report: 1 Volume
- Programs & Datasets: 1 Diskette

Abstract

Metal-ion binding proteins play important roles in biological processes. Metal-ion binding motifs have been identified for the common metal ions. However, some metal-ion binding proteins do not possess any canonical metal-ion binding motif. In this project, we investigate the hypothesis that there are sequence characteristics that are common to metal-ion binding proteins. Based on this hypothesis, we hope to develop a model for recognizing metal-ion binding proteins based on their sequence and physico-chemical parameters. Many metal-ion binding proteins commonly bind to divalent metal ions, but existing works say they possess unique motifs specific to that divalent ion. There is no accurate general fingerprints for metal-binding proteins. The establishment of such a general global fingerprint will help find novel metal-ion binding proteins.

Subject Descriptors:

J.3 Biology and genetics
I.5.2 Classifier design and evaluation
I.5.2 Feature evaluation and selection
I.5.3 Clustering

Keywords:

Metalloproteins, KD-Tree, Correlation-based Feature Subset Selection, C4.5 algorithm, Support Vector Machines

Implementation Software and Hardware:

Swiss-Prot, JRuby, Datamapper, SQLite, WEKA, ANN

Acknowledgement

I would like to thank Prof. Limsoon Wong for the insights into data-mining and Prof. Wing-Kin Ken Sung for the ideas from phylogenetics, both of which was essential for the completion of this project.

Table Of Contents

Introduction 5

Evolution of Metalloproteins 5

General Experimental Strategy & Rationale 6

Experiment 7

Protein Sequence Database 7

Feature Vector Creation 8

k-NN Calculation 9

Training/Test Set Creation 9

Feature Selection 9

Classifier Evaluation 9

Results 10

Conclusion 10

References 11

Appendix A - Program Listings 12

model.rb 12

populate_proteins.rb 13

populate_features.rb 13

populate_features_csv.rb 15

generate_weka_csv.rb 15

generate_all_csv.rb 15

pts2ids.rb 15

approximate_nearest_neighbor.cpp 16

Introduction

Throughout evolution, properties of metals have been harnessed by proteins for performing functions such as redox reactions which cannot be performed by using functional groups found amino acids.(Messerschmidt, Huber, Wieghardt and Poulos, 2001). These metalloproteins have many different functions in cells such as, enzymes, transport, storage, and signal transduction.

Experimental biologists use techniques such as QPNC-PAGE, ICP Mass Spectroscopy and NMR Spectroscopy, to isolate and classify metalloproteins from a mixture of proteins. A large amount of literature describing results of such experiments are now available, many of which are curated in protein sequence databases.

These databases show that 30% of all proteins are metalloproteins and that many pathways contain at least one metalloenzyme and require Mg, K, Ca, Fe, Mn and Zn to sustain life. Other elements like Cu, Mo, Ni, Se, and Co, are required by lesser organisms. The different metals have a range of affinities for most protein environments in or the order of Mg+2/Ca+2 < Mn+2 < Fe+2 < Co+2 < Ni+2 < Cu+2 ~Zn+2, an equilibrium series known as the Irving-Williams Series. (Dupont et al., 2010).

There have been previous successful attempts (Cai, Han, Ji, Chen and Chen, 2003) to use features calculated from sequences in protein databases to accurately classify proteins as specific metalloproteins.

This experiment aims to test whether there are any signatures common to all metalloproteins. In order to do so, a crucial question must be answered (at least partially): How did metalloproteins came out to be? The answer to this question will help us design better classifiers.

Evolution of Metalloproteins

Early ocean chemistry was dramatically different from today. Oxygen was absent and trace elements such as iron, manganese, and cobalt were abundant. Photosynthesis resulted in an abundance of oxygen around 2.4 billion years ago and the oceans started to accumulate oxygen,

increasing the amount of zinc, copper, and molybdenum that was available. At the same time, iron became very rare. (Dupont et al. 2010)

The first organisms predominantly used metals that were abundant in the ancient ocean, Fe, Mn, and Co. This metal utilization bias is preserved to this day in the Bacteria and Archaea, which still predominantly use ancient protein structures. Later, as the ocean accumulated oxygen, new proteins evolved that bound zinc and copper. So did the Eukaryotes, which include all organisms with a nucleus, from single-cell plankton to humans. (Dupont et al. 2010)

It is now known that the new zinc and copper-binding proteins are only found in Eukaryotes, not in the Bacteria and Archaea. The nucleus houses most of the new zinc binding proteins and this unique utilization of zinc is one of the defining features of all Eukaryotes. A possible hypothesis is that zinc concentrations in the ancient ocean were too low to allow for the evolution of the Eukaryotes, at least until global changes in oxygen occurred. (Dupont et al. 2010)

Since evolution of metalloproteins has happened multiple times under different selection pressures and for different metals, it can be hypothesized that any overarching signature common to all metalloproteins will be very weakly predictive.

Finally it can be safely argued that each metalloprotein shares a common ancestor with a non-metal-binding protein and that an alignment free distance metric such as Euclidean distance between *k*-mer frequency vectors (Edgar, 2004) could approximate the actual evolutionary distance between a metalloprotein and its nearest non-metal-binding relative. This fact will be useful for creating the training and test sets.

General Experimental Strategy & Rationale

A protein sequence database with minimal redundancy, maximal annotation and maximum correspondence with reality (i.e. maximum experimental evidence) is chosen. This so as to minimize the amount of computation necessary to verify these aspects of each database entry.

Next each protein sequence in the database is converted to a feature vector consisting of 104 dimensions (Han et. al, 2004). These dimensions contain encoded representations of tabulated

amino acid residue properties including amino acid composition, hydrophobicity, normalized Van der Waals volume, polarity, polarizability, charge, surface tension, secondary structure and solvent accessibility. (Han et. al, 2004). The exact details of this feature vector will be discussed in the following sections.

Since any classifier capable of identifying metalloproteins would also have to distinguish them from the neighboring non-metalloproteins, it is obvious that both the training set and the test set has to contain a set of metalloproteins and its nearest non-metal-binding neighbors in it. Furthermore since the part of the feature vector which describes the amino acid composition pays resemblance to *k*-mer frequency vectors (Edgar, 2004) the Euclidean distance between which approximates the actual evolutionary distance between proteins, this implies that a training/test set would contain metalloproteins and its nearest non-metal-binding evolutionary relatives. This is desirable because this will allow any supervised classifier to derive the exact nature of the evolutionary events which resulted in the incorporation of the metal-ions into proteins.

So *k*-nearest neighbors of each metalloprotein must be calculated. This is going to be challenging due to the high dimensionality of the feature vector. However as we shall see in the following sections, this problem can be solved by resorting to approximate solutions.

Once the training/test sets have been created as described above, a Correlation-based Feature Subset Selection (Hall, 1999) is used to select subsets of features that are highly correlated with the class while having low intercorrelation.

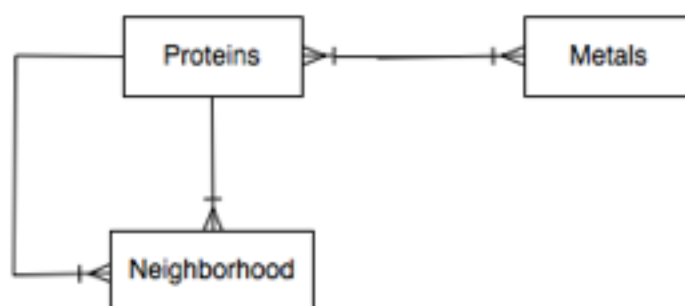
Next, a classifier such as a C4.5 tree or a support vector machine is used to extract patterns from the training set. This classifier is evaluated using cross-validation and other test sets.

Experiment

Protein Sequence Database

The protein sequence database chosen for this experiment was SwissProt (Bairoch and Apweiler, 2000). It is a manually curated, large, non-redundant sequence database available as a flat file which

could be accessed sequentially. So as to facilitate easy random access and manipulation, a relational



database based on SwissProt was created with a schema such as the one shown here.

Feature Vector Creation

First amino acids are organized into three different groups based on physio-chemical properties such as hydrophobicity, Van der Waals volume, Polarity and Polarizability. (Han et. al, 2004)

Property	Group 1	Group 2	Group 2
Hydrophobicity	RKEDQN (Polar)	GASTPHY (Neutral)	CVLIMFW (Hydrophobic)
Van der Waals Volume	GASCTPD (0-2.78)	NVEQIL (2.95-4.0)	MHKFRYW (4.43-8.08)
Polarity	LIFWCMVY (4.9–6.2)	PATGS (8.0–9.2)	HQRKNED (10.4–13.0)
Polarizability	GASDT (0–0.108)	CPNVEQIL (0.128–0.186)	KMHFRYW (0.219–0.409)

Table 1: Division of amino acids into three different groups for different physicochemical properties

For each property described in Table 1, three descriptors are calculated:

- Composition (C): C is the number of amino acids of a particular property (such as hydrophobicity) divided by the total number of amino acids in a protein sequence.
- Transition (T): T characterizes the percentage frequency with which amino acids of a particular property is followed by amino acids of a different property.
- Distribution (D): D measures the chain length within which the first, 25, 50, 75 and 100% of the amino acids of a particular property is located respectively.

Finally the 20 amino acid composition are also calculated and added to the feature vector. Together these properties add up to 104 dimensions, excluding class.

k-NN Calculation

Computing exact nearest neighbors in dimensions much higher than 8 seems to be a very difficult task. Few methods seem to be significantly better than a brute-force computation of all distances. However, it has been shown that (Arya et al., 1998) by computing nearest neighbors approximately, it is possible to achieve significantly faster running times (on the order of 10's to 100's) often with a relatively small actual errors.

Unfortunately even the approximate *k*-NN algorithm starts accumulating errors beyond 20 dimensions. So in order to circumvent this, 20 principal components which accounted for most of the standard deviation among all proteins where calculated. These 20 principal components were used as part of the approximate *k*-NN calculation.

40 non-metal-binding neighbors were calculated for each metalloprotein.

Training/Test Set Creation

For each metal binding protein feature vector in the training set, and equal number of non-metal-binding feature vectors were included.

A training set for Calcium, Copper, Magnesium, Manganese, Nickel, Sodium and Zinc were created containing all of the respective metal-binding proteins. An additional training file with all known metalloproteins and their nearest non-metal-binding proteins was also created.

Feature Selection

Since a 104 dimensional dataset is too sparse, Correlation-based feature selection was used to find subsets of features that are highly correlated with the class while having low intercorrelation.

Classifier Evaluation

Both C4.5 Trees and Support Vector Machines (with Polynomial Kernels) were used as classifiers. They were evaluated using a 10-fold cross validation.

Results

Here are the Precision, Recall and F-Measure for classifiers of different metalloproteins after a 10-fold cross validation:

Metal	Classifier	Precision	Recall	F-Measure
Sodium	C4.5	97.3%	97.3%	97.3%
Nickel	C4.5	91.9%	91.9%	91.9%
Copper	C4.5	85.3%	85.3%	85.3%
Iron	C4.5	83.9%	83.9%	83.9%
Manganese	C4.5	83.6%	83.6%	83.6%
Magnesium	C4.5	81%	81%	81%
Calcium	C4.5	78.3%	78.3%	78.3%
Zinc	C4.5	69.7%	69.6%	69.4%
All Metals	SMO (SVM)	65%	64.9%	63.6%

The above results indicate that the Sodium ion binding proteins are the most predictable from its feature vector, followed by Nickel, Copper Iron, Manganese, Magnesium, Calcium and Zinc.

Conducting a 10-fold cross validation of a classifier which was trained using a training set containing all known metalloproteins and their nearest non-metal-binding neighbors indicates that, there is indeed a signature that is common to all known metalloproteins. However as indicated by the evolutionary history of metalloproteins, this signature has a very weak predictive power.

Conclusion

This experiment indicates that a combination of C4.5 classifier and Correlation-based feature selection has great potential in detecting specific metal-binding protein classes especially Sodium and Nickel binding proteins. Furthermore it also indicates that there exists a general fingerprint common to all metalloproteins. However this signature is not very useful for practical use. Instead an ensemble of classifiers trained to recognize specific metalloprotein signatures would in effect behave like a general metalloprotein classifier with higher accuracies.

References

Arya et al. An optimal algorithm for approximate nearest neighbor searching fixed dimensions. Journal of the ACM (JACM) (1998) vol. 45 (6) pp. 891-923

Bairoch and Apweiler. The SWISS-PROT protein sequence database and its supplement TrEMBL in 2000. Nucleic Acids Research (2000) vol. 28 (1) pp. 45

Cai et al. SVM-Prot: web-based support vector machine software for functional classification of a protein from its primary sequence. Nucleic Acids Research (2003) vol. 31 (13) pp. 3692

Dupont et al. History of biological metal utilization inferred through phylogenomic analysis of protein structures. Proceedings of the National Academy of Sciences (2010) vol. 107 (23) pp. 10567

Edgar. Local homology recognition and distance measures in linear time using compressed amino acid alphabets. Nucleic Acids Research (2004) vol. 32 (1) pp. 380

Han et al. Predicting functional family of novel enzymes irrespective of sequence similarity: a statistical learning approach. Nucleic Acids Research (2004) vol. 32 (21) pp. 6437

Hall. Correlation-based feature selection for machine learning. Thesis submitted in partial fulfilment of the requirements of the degree of Doctor of Philosophy at the University of Waikato. (1999)

Hall et al. The WEKA data mining software: An update. ACM SIGKDD Explorations Newsletter (2009) vol. 11 (1) pp. 10-18

Messerschmidt, A; Huber, R.;Wieghardt,K.;Poulos, T. (2001). Handbook of Metalloproteins. Wiley

Appendix A - Program Listings

model.rb

```
require 'rubygems'
require 'dm-core'
require  'dm-migrations'
require  'dm-types'

DataMapper::Logger.new($stdout, :debug)
DataMapper.setup(:default, "sqlite://#{Dir.pwd}/proteins.db")

class Protein
  include DataMapper::Resource

  property :id, Serial
  property :entry_id, String, :required => true, :unique => true
  property :features, CommaSeparatedList, :default => '', :lazy => true
  property :features_csv, Text, :default => ''
  property :metals_count, Integer, :default => 0
  property :aaseq, Text, :required => true

  has n, :metals, :through => Resource
  has n, :family, :child_key => [ :a_id ]
  has n, :relatives, self, :through => :family, :via => :b
end

class Metal
  include DataMapper::Resource

  property :id, Serial
  property :name, String, :required => true, :unique => true
  has n, :proteins, :through => Resource
end

class Family
  include DataMapper::Resource

  property :a_id, Integer, :key => true, :min => 1
  property :b_id, Integer, :key => true, :min => 1
  property :nn, Integer

  belongs_to :a, 'Protein', :key => true
  belongs_to :b, 'Protein', :key => true
end

DataMapper.finalize
DataMapper.auto_upgrade!

metal_names = ["Cadmium", "Calcium", "Cobalt", "Copper", "Iron", "Lead", "Lithium",
"Magnesium", "Manganese", "Mercury", "Molybdenum", "Nickel", "Potassium", "Selenium",
"Sodium", "Tungsten", "Vanadium", "Zinc"]

if Metal.count == 0
  metal_names.each { |name|
    Metal.create(:name => name)
  }
end
```

populate_proteins.rb

```
require 'rubygems'
require 'dm-core'
require 'bio'

require 'models'

metal_names = ["Cadmium", "Calcium", "Cobalt", "Copper", "Iron", "Lead", "Lithium",
"Magnesium", "Manganese", "Mercury", "Molybdenum", "Nickel", "Potassium", "Selenium",
"Sodium", "Tungsten", "Vanadium", "Zinc"]

metals = {}
metal_names.each { |name|
  metals[name] = Metal.first(:name => name)
}

sequences = Bio::FlatFile.auto(ARGF)

sequences.each do |seq|
  protein = Protein.new(:entry_id => seq.entry_id, :aaseq => seq.aaseq)
  if seq.kw.include?("Metal-binding")
    m = (metal_names & seq.kw)
    m.each { |name| protein.metals << metals[name]}
    protein.metals_count = m.count
  end
  protein.save
end
```

populate_features.rb

```
require 'rubygems'
require 'dm-core'
require  'dm-types'
require 'models'

Features = [
  #Composition
  #Hydrophobicity
  /[RKEDQN]/,
  /[GASTPHY]/,
  /[CVLIMFW]/,
  #Van der Walls volume
  /[GASCTPD]/,
  /[NVEQIL]/,
  /[MHKFRYW]/,
  #Polarity
  /[LIFWCMVY]/,
  /[PATGS]/,
  /[HQRKNED]/,
  #Polarizability
  /[GASDT]/,
  /[CPNVEQIL]/,
  /[KMHFRYW]/,

  #Transition
  #Hydrophobicity
  /[RKEDQN][GASTPHY]/,
  /[GASTPHY][RKEDQN]/,
  /[RKEDQN][CVLIMFW]/,
  /[CVLIMFW][RKEDQN]/,
  /[GASTPHY][CVLIMFW]/,
  /[CVLIMFW][GASTPHY]/,
  #Van der Walls volume
```

```
  /[GASCTPD][NVEQIL]/,
  /[NVEQIL][GASCTPD]/,
  /[GASCTPD][MHKFRYW]/,
  /[MHKFRYW][GASCTPD]/,
  /[NVEQIL][MHKFRYW]/,
  /[MHKFRYW][NVEQIL]/,
  #Polarity
  /[LIFWCMVY][PATGS]/,
  /[PATGS][LIFWCMVY]/,
  /[LIFWCMVY][HQRKNED]/,
  /[HQRKNED][LIFWCMVY]/,
  /[PATGS][HQRKNED]/,
  /[HQRKNED][PATGS]/,
  #Polarizability
  /[GASDT][CPNVEQIL]/,
  /[CPNVEQIL][GASDT]/,
  /[GASDT][KMHFRYW]/,
  /[KMHFRYW][GASDT]/,
  /[CPNVEQIL][KMHFRYW]/,
  /[KMHFRYW][CPNVEQIL]/
]

def distr(seq)
  l=seq.length.to_f
  dist= Hash.new(0.0)
  seq.each_char do |aa|
    dist[aa] += 1
  end
  dist.each_key do |aa|
    dist[aa]/=l
  end
  return dist.values_at("A", "C", "D", "E", "F", "G", "H", "I", "K", "L", "M", "N", "P",
"Q", "R", "S", "T", "V", "W", "Y")
end

def gen_vector(seq)
  c = []
  t_detail = []
  t = []
  d = []
  Features.each_index { |i|
    if i < 12
      c << seq.scan(Features[i]).length.to_f/seq.length.to_f
      matches = seq.enum_for(:scan, Features[i]).map {
       Regexp.last_match.begin(0) + 1
      }
      d << matches.first.to_f/seq.length.to_f
      d << matches[matches.length/4 - 1].to_f/seq.length.to_f
      d << matches[matches.length/2 - 1].to_f/seq.length.to_f
      d << matches[3*matches.length/4 - 1].to_f/seq.length.to_f
      d << matches.last.to_f/seq.length.to_f
    else
      t_detail << seq.scan(Features[i]).length.to_f/(seq.length.to_f - 1)
    end
  }
  t_detail.each_slice(2) { |slice|
    t << slice[0] + slice[1]
  }
  return distr(seq) + c + t + d
end
(Protein.first.id..Protein.last.id).each { |i|
  puts i
  protein = Protein.get(i)
  protein.features = gen_vector(protein.aaseq)
  protein.save
}
```

populate_features_csv.rb

```
require 'rubygems'
require 'models'

(Protein.first.id..Protein.last.id).each { |i|
  puts i
  protein = Protein.get(i)
  protein.features_csv = protein.features.join(",")
  protein.save
}
```

generate_weka_csv.rb

```
require 'models'

file = File.new("#{Dir.pwd}/#{ARGV.first}.csv", "w")
proteins = Protein.all(:metals_count => 1,Protein.metals.name => ARGV.first)

relatives = []
header = ["A1"]
104.times { header << header.last.succ }
file.puts header.join(",")
proteins.each { |protein|
  file.puts protein.features.join(",") + ",METAL"
  relative = protein.relatives.first
  puts protein.entry_id if relative == nil
  if !relatives.include?(relative.id)
    file.puts relative.features.join(",") + ",NONMETAL"
    relatives << relative.id
  end
}
```

generate_all_csv.rb

```
require 'rubygems'
require 'dm-core'
require  'dm-types'
require 'models'
require 'fastercsv'

metal_file = File.new("#{Dir.pwd}/metal.csv", "w")
non_metal_file = File.new("#{Dir.pwd}/non_metal.csv", "w")
(Protein.first.id..Protein.last.id).each { |i|
  puts i
  protein = Protein.get(i)
  metal_file.puts protein.entry_id + "," + protein.features.to_csv if
protein.metals_count == 1
  non_metal_file.puts protein.entry_id + "," + protein.features.to_csv if
protein.metals_count == 0
}
```

pts2ids.rb

```
#Converts points returned by ANN program into Protein IDs.
require 'models'
nn = File.new("40_nn.csv", "r")
metal = File.new("metal.csv", "r")
non_metal = File.new("non_metal.csv", "r")
output = File.new("40_nn_ids.csv","w")

metal_ids = []
non_metal_ids = []
```

```
while(metal_line = metal.gets)
  metal_ids << Protein.first(:entry_id => metal_line.split(",")[0]).id
end

while(non_metal_line = non_metal.gets)
  non_metal_ids << Protein.first(:entry_id => non_metal_line.split(",")[0]).id
end

output.puts "A_ID,B_ID,NN"
while(nn_line = nn.gets)
  nn_array = nn_line.split(",")
  m = metal_ids[nn_array[0].to_i]
  nm = non_metal_ids[nn_array[2].to_i]
  output.puts "#{m},#{nm},#{nn_array[1]}"
end
```

approximate_nearest_neighbor.cpp

```
#include <cstdlib>                                          // C standard library
#include <cstdio>                                           // C I/O (for sscanf)
#include <cstring>                                          // string manipulation
#include <fstream>                                          // file I/O
#include <ANN/ANN.h>                                // ANN declarations

using namespace std;                                // make std:: accessible

//----------------------------------------------------------------------
// ann_sample
//
// This is a simple sample program for the ANN library.     After compiling,
// it can be run as follows.
//
// ann_sample [-d dim] [-max mpts] [-nn k] [-e eps] [-df data] [-qf query]
//
// where
//            dim                         is the dimension of the space (default = 2)
//            mpts                 maximum number of data points (default = 1000)
//            k                           number of nearest neighbors per query (default 1)
//            eps                         is the error bound (default = 0.0)
//            data                 file containing data points
//            query                file containing query points
//
// Results are sent to the standard output.
//----------------------------------------------------------------------

//----------------------------------------------------------------------
//     Parameters that are set in getArgs()
//----------------------------------------------------------------------
void getArgs(int argc, char **argv);                // get command-line arguments

int                         k                             = 1;    // number of nearest neighbors
int                         dim                           = 2;    // dimension
double              eps                           = 0;            // error bound
int                         maxPts                = 1000;         // maximum number of data points

istream*            dataIn                = NULL;                 // input for data points
istream*            queryIn               = NULL;                 // input for query points

bool readPt(istream &in, ANNpoint p)                // read point (false on EOF)
{
       for (int i = 0; i < dim; i++) {
              if(!(in >> p[i])) return false;
       }
       return true;
}

void printPt(ostream &out, ANNpoint p)                      // print point
{
       out << "(" << p[0];
       for (int i = 1; i < dim; i++) {
```

```
		out << ", " << p[i];
	}
	out << ")\n";
}

int main(int argc, char **argv)
{
	int				nPts;			// actual number of data points
	ANNpointArray		dataPts;			// data points
	ANNpoint			queryPt;			// query point
	ANNidxArray			nnIdx;			// near neighbor indices
	ANNdistArray		dists;			// near neighbor distances
	ANNkd_tree*			kdTree;			// search structure

	getArgs(argc, argv);					// read command-line arguments

	queryPt = annAllocPt(dim);				// allocate query point
	dataPts = annAllocPts(maxPts, dim);			// allocate data points
	nnIdx = new ANNidx[k];					// allocate near neigh indices
	dists = new ANNdist[k];						// allocate near neighbor
dists

	nPts = 0;								// read data points

	//cout << "Data Points:\n";
	while (nPts < maxPts && readPt(*dataIn, dataPts[nPts])) {
		//printPt(cout, dataPts[nPts]);
		nPts++;
	}

	kdTree = new ANNkd_tree(					// build search structure
				dataPts,			// the data points
				nPts,				// number of points
				dim);				// dimension of space

  int q = 0; //Query Point Number

  cout << "METAL,ORDER,NONMETAL,DISTANCE\n";
	while (readPt(*queryIn, queryPt)) {			// read query points
		//cout << "Query point: ";			// echo query point
		//printPt(cout, queryPt);

		kdTree->annkSearch(				// search
			queryPt,				// query point
			k,					// number of near neighbors
			nnIdx,					// nearest neighbors (returned)
			dists,					// distance (returned)
			eps);					// error bound

		for (int i = 0; i < k; i++) {			// print summary
			dists[i] = sqrt(dists[i]);		// unsquare distance
			cout << q << "," << i << "," << nnIdx[i] << "," << dists[i] << "\n";
		}
    q++;
	}
    delete [] nnIdx;						// clean things up
    delete [] dists;
    delete kdTree;
	annClose();							// done with ANN

	return EXIT_SUCCESS;
}

//----------------------------------------------------------------------
//	getArgs - get command line arguments
//----------------------------------------------------------------------

void getArgs(int argc, char **argv)
{
	static ifstream dataStream;				// data file stream
	static ifstream queryStream;				// query file stream
```

```
	if (argc <= 1) {						// no arguments
		cerr << "Usage:\n\n"
		<< "  ann_sample [-d dim] [-max m] [-nn k] [-e eps] [-df data]"
		   " [-qf query]\n\n"
		<< "  where:\n"
		<< "    dim      dimension of the space (default = 2)\n"
		<< "    m        maximum number of data points (default = 1000)\n"
		<< "    k        number of nearest neighbors per query (default 1)\n"
		<< "    eps      the error bound (default = 0.0)\n"
		<< "    data     name of file containing data points\n"
		<< "    query    name of file containing query points\n\n"
		<< " Results are sent to the standard output.\n"
		<< "\n"
		<< " To run this demo use:\n"
		<< "    ann_sample -df data.pts -qf query.pts\n";
		exit(0);
	}
	int i = 1;
	while (i < argc) {					// read arguments
		if (!strcmp(argv[i], "-d")) {			// -d option
			dim = atoi(argv[++i]);			// get dimension to dump
		}
		else if (!strcmp(argv[i], "-max")) { // -max option
			maxPts = atoi(argv[++i]);		// get max number of points
		}
		else if (!strcmp(argv[i], "-nn")) {		// -nn option
			k = atoi(argv[++i]);			// get number of near neighbors
		}
		else if (!strcmp(argv[i], "-e")) {		// -e option
			sscanf(argv[++i], "%lf", &eps);		// get error bound
		}
		else if (!strcmp(argv[i], "-df")) {		// -df option
			dataStream.open(argv[++i], ios::in);// open data file
			if (!dataStream) {
				cerr << "Cannot open data file\n";
				exit(1);
			}
			dataIn = &dataStream;			// make this the data stream
		}
		else if (!strcmp(argv[i], "-qf")) {		// -qf option
			queryStream.open(argv[++i], ios::in);// open query file
			if (!queryStream) {
				cerr << "Cannot open query file\n";
				exit(1);
			}
			queryIn = &queryStream;			// make this query stream
		}
		else {							// illegal syntax
			cerr << "Unrecognized option.\n";
			exit(1);
		}
		i++;
	}
	if (dataIn == NULL || queryIn == NULL) {
		cerr << "-df and -qf options must be specified\n";
		exit(1);
	}
}
```