

APPENDIX B: IMPLEMENTATION

All code for this thesis was written in SQL and the Matlab programming language. An MS Access database was used for data storage and retrieval, and Matlab v6.1 was used to run all statistical evaluations and generate most figures. The ODBC standard and the Database Access Toolbox allowed for the transfer of data between these two platforms.

Given the large amount of code written for this thesis, it would be impossible to include a full compendium. Instead, the author has included particularly important sections of code (or pseudo-code, if it is easier). In the code I reference many functions, only a few of which are included. The ones included will be marked with **bold** lettering. Confusing sections of Matlab code that are particularly syntax-specific have been removed, as have various error catching mechanisms, etc. For this reason, some of the code will not run as-is.

B.1 Overview

This appendix is broken down into sections, corresponding to chapters in the body of the thesis. Most of the modules are self-sufficient and easy to comprehend. However, it is necessary that the reader have some understanding of what is done in SQL and what is done in Matlab. For this reason, below I give a pseudo-code overview of the steps involved in shotgun clustering. A quick read of this section will make the rest of the code in this appendix significantly more transparent.

Code B1.1: *Pseudo-code for general program flow (shotgun clustering)*

- (1) Connect to database. The database access toolbox allows Matlab to select and insert data over an ODBC connection. SQL queries are written as strings in Matlab.
- (2) Define an SQL Query to retrieve information about how to conduct shotgun clustering. No important information is stored in Matlab or in memory. All information that needs to be retained is entered into the database. In this case, the data retrieved is global “trial” parameters (e.g. which patients to include, which algorithms to use).
- (3) Execute SQL query and store the results in Matlab arrays and matrices. This is the data structure that defines a “trial”:

```
tdat = {Include_Genes, Set_Clustering, Patients, Genes, Condensation,  
Min_Size, Use_Shot_H, Use_Shot_K, Kvals, Min_Condensation}
```

- (4) Process this data structure in Matlab. Within the `tdat` structure is all the necessary information to do shotgun clustering; however, many of the field values are actually function calls. For instance, `genes` is a structure that stores executable matlab code (linked to the `GENES` field of the `TRIALS` table). This executable code is typically a call to `filtergenes` or `randomgenes`, which will in turn retrieve `GENE_IDS` from the `GENES` table that meet certain criteria.
- (5) Using information in `tdat`, construct a SQL query to retrieve the $m \times n$ matrix X and store it in Matlab.
- (6) Run shotgun clustering (`do_shotgenes`) on this matrix.
- (7) Convert the results into unique rows that will be inserted into the table `CLUSTER_RESULTS` with `ITERATION=1` and insert the rows into the database.

B.2 Database Schema and Code

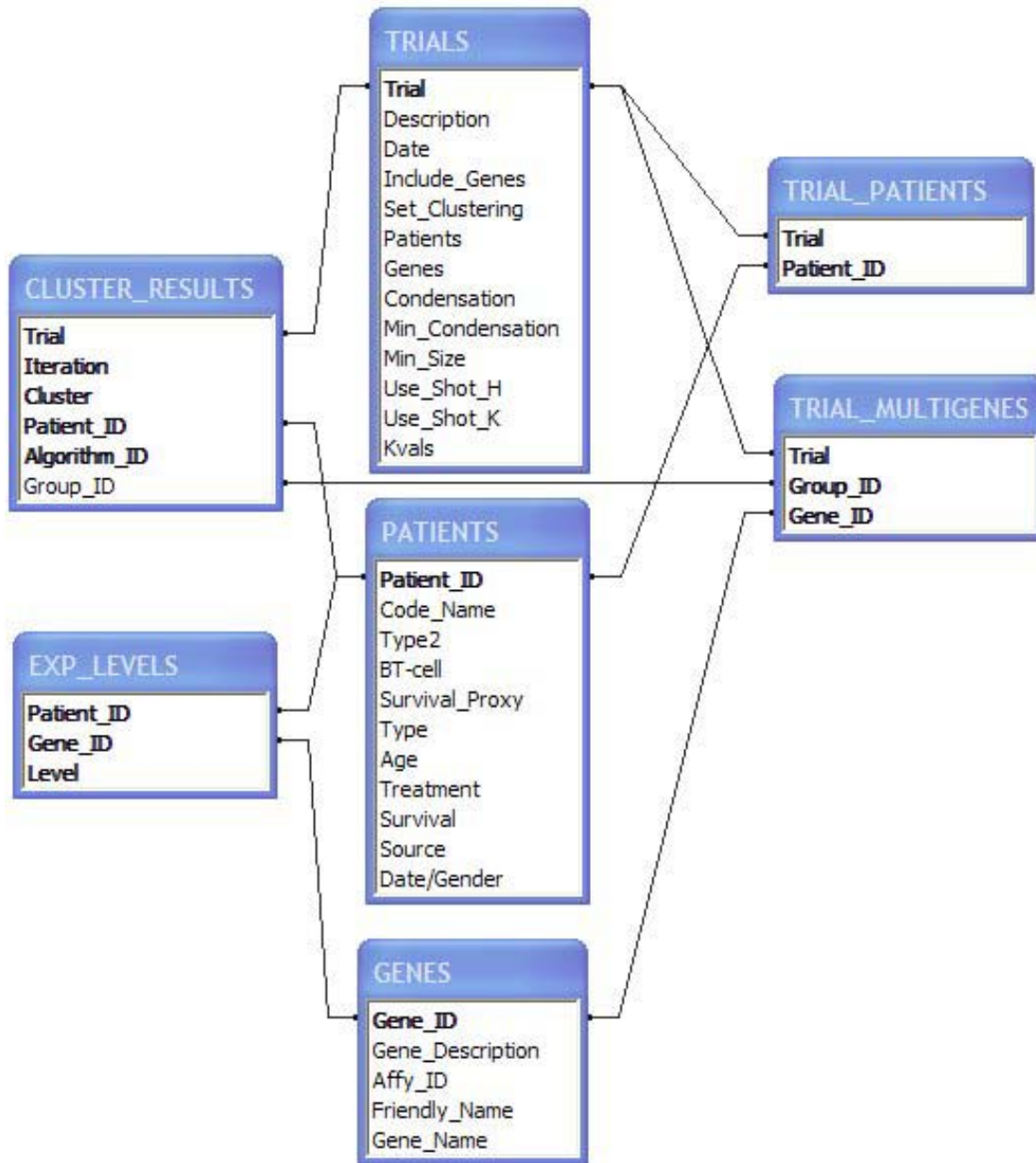


Figure B.1: Database Schema. Primary keys are indicated in bold.

Code B2.1: Typical Database Retrieval Function

```
function cr = getsetclusters(trial, iteration)

% Read collection  $I_i$  from CLUSTER_RESULTS table
% cr is an array of cells with fields <algs> and <pats>

logintimeout(10);
conn = database('buenodata', '', '');
setdbprefs({'DataReturnFormat', 'ErrorHandling'},
          {'numeric', 'report'});

selectQ = 'SELECT Patient_ID, Algorithm_ID, Cluster';
fromQ = 'FROM CLUSTER_RESULTS';
whereQ = [' WHERE Trial = ', num2str(trial), ...
         ' AND Iteration = ', num2str(iteration)];

query = [selectQ, fromQ, whereQ];
curs = exec(conn, query);
curs = fetch(curs);

raw_dat = curs.Data;
[num_rows, three] = size(raw_dat);

% allocate space in the list
maxes = max(raw_dat);
num_clus = maxes(1,3);
cr{num_clus} = [];

for r = 1:num_rows
% raw_dat(r,1) is the patient_ID
% raw_dat(r,3) is the clus # for the algorithm raw_dat(r,2)
    if isstruct(cr{raw_dat(r,3)}) == 1
        cr{raw_dat(r,3)}.pats = [cr{raw_dat(r,3)}.pats, raw_dat(r,1)];
        cr{raw_dat(r,3)}.algs = [cr{raw_dat(r,3)}.algs, raw_dat(r,2)];
    else
        cr{raw_dat(r,3)}.pats = raw_dat(r,1);
        cr{raw_dat(r,3)}.algs = raw_dat(r,2);
    end
end

close(curs);
close(conn);
clear curs;
```

Code B2.2: Typical Database Export Function

```
function exp2db(myData, trial, iteration)

% write <myData> to the CLUSTER_RESULTS table.
% <myData> must be of the form
% [Patient_ID, Algorithm_ID, Cluster, Group_ID].
% <Trial> and <Iteration> must be defined such that duplicate records
% will not be created in the CLUSTER_RESULTS table. Otherwise, a
% General error will occur in database/exec

colnames = {'Patient_ID', 'Algorithm_ID', 'Cluster', 'Group_ID',
            'Trial', 'Iteration'};

conn = database('buenodata', '', '');
setdbprefs({'DataReturnFormat', 'ErrorHandling'},...

% concatenate the trial and iteration information
tr_it=[];
[a,b] = size(myData);

for i=1:a
    tr_it = [tr_it;trial, iteration];
end
newData = [myData, tr_it];

insert(conn, 'CLUSTER_RESULTS', colnames, newData);
close(conn);
```

B.3 Shotgun Stage

Code B3.1: Top-level Shotgun Clustering Delegation Function

```
function s = do_shotgenes(trial, minclustersize, expats, genesets, ...
    kval, useH, useK)

% based on values in TRIALS table (retrieved by fcn run_trial), this
% delegates clustering to the appropriate functions

if (gettrialexists(trial)==1)
    ['trial already exists. exiting...'], return;
end

% update TRIAL_PATIENTS table
settrialpats(trial, expats);

% update TRIAL_GENES table
ngenesets = length(genesets)
for i=1:ngenesets
    settrialmultigenes(trial, i, genesets{i});
end
numgroups = gettrialgenegroups(trial)

% Run shotgun clustering, m is a strange Matlab structure
[m, groups] = shot_bygenes(trial, kval, useH, useK);

pats = gettrialpats(trial);
% convert m into a structure for insertion into CLUSTER_RESULTS
cr = conv_h(m,pats, minclustersize, groups);

% add data to database
exp2db(cr, trial, 1);
```

Code B3.2: Matlab code to filter genes

```
function l = sgenessubsets(condition, c1, c2);

% sgenessubsets uses a helper function SELECTGENES that dynamically
% creates a SQL query to select the genes meeting condition(c1, c2)
% for example, if condition=='avgbetween', SQL is:
%   SELECT EXP_LEVELS.Gene_ID FROM EXP_LEVELS
%   GROUP BY EXP_LEVELS.Gene_ID
%   HAVING (Avg(EXP_LEVELS.Level)> c1 AND (Avg(EXP_LEVELS.Level)< c2
% possible parameters to selectgenes are:
%   min, max, between, stdabove, stdbelow, stdbetween,...

if nargin==2
    geneIDs = selectgenes(condition, c1);
elseif nargin==3
    geneIDs = selectgenes(condition, c1, c2);
end

ngenes = length(geneIDs);
% divide all genes into subsets of these cardinalities
subsetsizes=[10, 50, 500];

% make sure we have enough genes to fill all sets
if max(subsetsizes)>ngenes,
    error('SGENESSUBSETS: not enough genes!'), return;
end

for i=1:length(subsetsizes)
%   generate a random list of indices
    rlist = randperm(ngenes);
%   allocate memory
    mycardinality = subsetsizes(1,i);
    rgenelist=zeros(1,mycardinality);
%   use rlist as random indices into geneIDs
    for j=1:mycardinality
        rgenelist(1,j)=geneIDs(rlist(j),1);
    end
%   store the answer and loop
    l{i}=rgenelist;
end
```

B.4 Consistency and Prevalence

Code B4.1: Pseudo-Code for calculating $\lambda(\Omega)$

```
//Step 1: calculate  $|\mathbf{U}(\Omega)|$ 
//use bits to "check off" each non-unique cluster
bits = array of ones of length  $|\Omega|$ 
for i= 1 to  $|\Omega|$  {
    //check to see if  $C_i$  is known to be non-unique
    if bits[i]==0{
        continue (to i=i+1);
    }
    else{
        for j = (i+1) to  $|\Omega|$  {
            if bits[j]==1 &&  $C_i==C_j$  {
                //Ci and Cj are not unique
                bits[i]=0;
                bits[j]=0;
            }
        }
    }
}

 $|\mathbf{U}(\Omega)|$  = sum(bits);

 $\lambda(\Omega) = \frac{|\mathbf{U}(\Omega)|}{|\Omega|}$ , as defined
```

B.5 Condensation Stage

Code B5.1: Main Condensation Clustering Routine

```
function s = do_shotgenes(trial, minclustersize,  $\tau$ ,
    maxits, setclustype, minCondensation)

% retrieve shotgun clusters from the CLUSTER_RESULTS table
s{1} = getshotclusters(trial);

% calculate centroids, if necessary
if (setclustype(1)=='c')
    genes = sort(gettrialmultigenesintersection(trial));
    [dat, pats] = getdata(pats, genes);
end

% multiset clustering occurs maxits times, or until convergence
for i=1:maxits
% based on value in TRIALS table, do appropriate multiset clustering
    switch setclustype
        case 'kmeans'
            sclus = ksetclus(s{i},  $\tau$ );
        case 'heirarchical'
            sclus = hsetclus(s{i},  $\tau$ );
        case 'centroid'
            sclus = csetclus(s{i},  $\tau$ , dat, pats);
        otherwise
            error('unrecognized setcluster type');
    end

% store results in database
    exp2db(sclus, trial, (i+1));

% retrieve results from database (same form as getshotclusters)
    s{i+1} = getsetclusters(trial, (i+1));

    if (length(s{i+1})<3)
% don't reduce to fewer than two clusters
        return;
    end
end
```

Code B5.2: Multiset Hierarchical Clustering

```
function cluster_results = hsetclus(s,  $\tau$ )

% setclus transforms a list of clusters into a matrix with columns
% [Patient_ID, Algorithm_ID, Cluster, Group_ID]
% The input s is the data structure generated by getclusterresults,
% i.e. it is an array of structs with fields <pats> and <alg>
% The second input is the condensation factor  $\tau$ 

numrows = length(s);

% use  $\mathcal{D}$  to calculate dissimilarity matrix
sdist = calcsetdists(s);

% use (e.g.) Ward linkage to generate hierarchical structure
slink = linkage(sdist, 'ward');

k_value = floor(numrows/ $\tau$ );
if (k_value<2)
    [' can't have fewer than 2 clusters, using 2...']
    k_value = 2;
end

% break dendrogram at the point necessary to create k_value clusters
cl = cluster(slink,k_value);

% convert output so it can be exported into dbase with exp2db
cluster_results = [];

for i = 1:numrows
% every patient in row i of s is in the cluster denoted by cl[i]
    clus = cl(i, 1);
    numpats = length(s{i}.pats);
    for p = 1:numpats
        cluster_results =
            [cluster_results; s{i}.pats(1,p), s{i}.algs(1,p), clus, -1];
    end
end
end
```

Code B5.3: Pseudo-code to calculate $\mathcal{D}(\mathbb{C}_p, \mathbb{C}_q)$

$\mathcal{D}=0$

for $\alpha=1$ to m {

$$\mathcal{D} = \mathcal{D} + \text{abs}\left(\frac{\text{card}(\mathbb{C}_p, X_\alpha)}{|\mathbb{C}_p|} - \frac{\text{card}(\mathbb{C}_q, X_\alpha)}{|\mathbb{C}_q|}\right)$$

}

return $\mathcal{D}/2$

B.6 Results & Visualization

Code B6.1: Matlab code to randomly rearrange a matrix (for Rand1)

```
function shuffledX = reshuffle(X, ntimes)

[m,n]=size(X);
for i=1:ntimes
    % reorder each row one at a time
    for r=1:m
        X(r,:) = X(r, randperm(n));
    end
    % reorder each column one at a time
    for c=1:n
        X(:,c) = X(randperm(m),c);
    end
end
shuffledX = X;
```

Code B6.2: Matlab code to randomly generate sets (for Rand2)

```
function parts = randompartitions(kvals, numperkval)

% for each kval in kvals, returns numperkval different partitions
% retrieve the list of all patients from PATIENTS table
pats = gettrialpats(3);
npats = length(pats);

cnt=0;
for k=kvals
    for npk=1:numperkval
        % memory allocation stuff
        clear rs;
        rs{k} = [];
        % randomly assign each patient to a cluster
        for p = pats
            c = ceil(rand*k);
            rs{c} = [rs{c}, p];
        end
        % add the new partition (rs) to the list of partitions
        for i=1:length(rs)
            cnt = cnt+1;
            parts{cnt}.pats = rs{i};
        end
    end
end
end
```

Code B6.3: Calculate normalized multisets and display heatmap

```
function [mems, pats] = memberships(trial, iteration, memtype, graphtype)

% returns a matrix of percent memberships, and a list of patients
% if graphtype = 'colormap', or 'mesh', or 'surf', displays graph
% if memtype = 'bypatient', normalization is by column
% if memtype = 'bycluster', normalization is by row

% get a list of the relevant patients
pats = gettrialpats(trial);
npats = length(pats);

% get a list of the relevant algorithms
algs = getalgs(trial);
nalgs = length(algs);

% retrieve data from CLUSTER_RESULTS
s = getclusters(trial, iteration);
nclus_it = length(s);

mems(npats,nclus_it) = 0;

switch memtype
case 'bypatient'
    for pat = pats
        pcount = patcount(trial, pat);
        for clus = 1:nclus_it
            mems(pat, clus) =
                (numoccurrences(pat, s{clus}.pats) / pcount);
        end
    end
case 'bycluster'
    for clus = 1:nclus_it
        clength = length(s{clus}.pats);
        for pat = pats
            mems(pat, clus) =
                (numoccurrences(pat, s{clus}.pats) / clength);
        end
    end

if (nargin>3)
% make the heatmap using the matrix just calculated (mems)
switch graphtype
case 'colormap'
    [m,n] = size(mems);
% the weird stuff adds a dummy row on the top and right
pcolor(1:(n+1),1:(m+1),[mems, zeros(m,1); zeros(1,n+1)]);
shading flat;
colorbar;
case 'mesh'
    [m,n] = size(mems);
    mesh(1:n,1:m,mems);
case 'surf'
    [m,n] = size(mems);
    surf(1:n,1:m,mems);
end
end
```