GENETIC ALGORITHMS

FOR

SAMPLE CLASSIFICATION OF MICROARRAY DATA

A Thesis

Presented to

The Graduate Faculty of The University of Akron

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

Dongqing Liu

December, 2005
GENETIC ALGORITHMS
FOR
SAMPLE CLASSIFICATION OF MICROARRAY DATA

Dongqing Liu

Thesis

Approved:

Co-Advisor
Zhong-Hui Duan

Accepted:

Dean of the College
Roger B. Creel

Co-Advisor
Jianping Zhu

Dean of the Graduate School
George R. Newkome

Faculty Reader
Yingcai Xiao

Date

Department Chair
Wolfgang Pelz
ABSTRACT

Many cancers consist of several different subtypes. For example, the two most common subtypes of renal cell carcinoma (RCC) are the clear cell RCC and the papillary RCC. It is expected that the gene expression profiles of the subtypes of common cancers are also distinctive and the subtypes can be identified based on the expressions of a panel of genes.

The goal of this thesis is to identify the panel of discriminator genes using a genetic algorithm and the k-nearest neighbor method. The genetic algorithm implemented uses integer-stream coding scheme. The fitness of each chromosome is evaluated by its ability to correctly classify the known samples using a k-nearest neighbor method.

To test the robustness of the algorithm, a bootstrapping analysis is performed, which removes one sample from the data set at a time and uses the remaining samples for gene selection. The effects of different distance metrics on the classification results, the stability of the algorithm with respect to different initial populations, and the sensitivity of the algorithm with respect to different samples are also studied.

The algorithm has been tested using two microarray data sets: a set of nine RCC samples and a set of 73 human acute leukemia samples. The computation results indicate the combined genetic algorithm and k-nearest neighbor method can serve as an effective tool for classifying cancer subtypes.
ACKNOWLEDGEMENTS

I would like to express my sincere appreciation to my advisors, Dr. Zhong-Hui Duan and Dr. Jiangping Zhu, for their constant encouragement and invaluable guidance during this study and other research projects. I am grateful to them for offering the opportunity to work in their research group. I am very impressed by their kindness and personality. This thesis and my study in Computer Science Department would not have been possible without their help and supports.

I would like to acknowledge the help of Computer Science Department for offering me the assistantship. I would like to acknowledge the help from Dr. Wolfgang Pelz, Dr. Yingcai Xiao, Dr. Timothy W. O’Neil, Dr. Xuan-Hien Dang, Dr. C.C. Chan and Ms. Peggy Speck for their constant assistance.

I would like to dedicate this thesis to my parents, my husband and my long-separate son. Without their encouragement, love and supports, I do not think I can finish this degree, this thesis and the study at the University of Akron. I am forever indebted to them, for the sacrifices they make to help me to achieve this success.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>LIST OF TABLES</th>
<th>vii</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIST OF FIGURES</td>
<td>viii</td>
</tr>
</tbody>
</table>

## CHAPTER

### I. INTRODUCTION

1

### II. GENETIC ALGORITHM

6

2.1 Introduction of Genetic Algorithm 
6

2.2 Chromosome Representation
8

2.3 Fitness Function / Cost Function
9

2.3.1 K-Nearest Neighbor (KNN) Method
10

2.3.2 KNN Performance and Drawbacks
11

2.3.3 Distance Measures
11

2.4 Initial Population
12

2.4.1 Pre-selection
12

2.5 Genetic Operation
13

2.5.1 Crossover
13

2.5.1.1 Paring
14

2.5.1.1.1 Pairing from Top to Bottom
15

2.5.1.1.2 Random Pairing
15
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>24</td>
</tr>
<tr>
<td>3.2</td>
<td>25</td>
</tr>
<tr>
<td>3.3</td>
<td>27</td>
</tr>
<tr>
<td>3.4</td>
<td>29</td>
</tr>
<tr>
<td>3.5</td>
<td>30</td>
</tr>
<tr>
<td>3.6</td>
<td>31</td>
</tr>
<tr>
<td>3.7</td>
<td>33</td>
</tr>
<tr>
<td>3.8</td>
<td>34</td>
</tr>
</tbody>
</table>
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Typical GA Cycle</td>
<td>7</td>
</tr>
<tr>
<td>2.2</td>
<td>Chromosome Representation</td>
<td>9</td>
</tr>
<tr>
<td>2.3</td>
<td>Crossover Operation</td>
<td>13</td>
</tr>
<tr>
<td>2.4</td>
<td>GA Algorithm Flowchart</td>
<td>19</td>
</tr>
<tr>
<td>3.1</td>
<td>Gene Ids and Number of Times Selected</td>
<td>26</td>
</tr>
<tr>
<td>3.2</td>
<td>Top 50 Genes Frequency for Dataset One</td>
<td>26</td>
</tr>
<tr>
<td>3.3</td>
<td>Top 50 Genes Frequency for Dataset Two</td>
<td>32</td>
</tr>
<tr>
<td>3.4</td>
<td>Gene Ids and Number of Times Selected for Dataset Two</td>
<td>35</td>
</tr>
</tbody>
</table>
CHAPTER I
INTRODUCTION

A DNA microarray is an orderly arrangement of tens to hundreds of thousands of DNA fragments (probes) of known sequences. It provides a platform for probe hybridization to radioactive or fluorescent labeled cDNAs (targets). The intensity of the radioactive or fluorescent signals generated by the hybridization reveals the level of the cDNAs in the biological samples under study. Using this new technology, distinct experiments with different tissues, patients, developmental stages, treatments, and times after a treatment etc. can provide different gene profiles under the varied experimental conditions studied. This gives the researchers a far better picture of the interactions among thousands of genes simultaneously, rather than the traditional ‘one gene in one experiment’ base.

DNA microarray data consists of a huge matrix or matrices; each has thousands of rows, representing the gene expressions used in the study, and ten to several hundred columns corresponding to the different samples. Alternately, DNA microarrays can sometimes consist of the gene expressions measured under different experimental conditions. However, while the number of genes may be great, the number of ways they behave is much more limited; the gene expression data is highly redundant, and most of the genes are not informative. Furthermore, in the microarray the signal to noise ratios tend to be low, thus increasing the difficulty of retrieving useful information.
DNA microarray technology is having a significant impact on genomics study, including drug discovery and toxicological research. Microarrays have become valuable tools for studying the gene expression patterns of both normal and diseased tissues. Class prediction and feature selection are the keys in the context of diagnostic applications of DNA microarrays.

Classification and prediction are two forms of data analysis; classification can be used to extract models that describe important data classes, while prediction can be used to extrapolate future data trends. The best-known approaches for data classification include decision tree induction, Bayesian classification and neutral networks; some other approaches are k-nearest neighbor classification, case-based reasoning, genetic algorithms, and fuzzy logic techniques. Data classification is a two-step process [7]. In the first step, a model is built by using a training sample; this model is used to describe a predetermined set of data classes. This is a supervised learning procedure, as the class label of each training sample has been provided. In the second step, the model is used to classify a testing dataset; at this time, some techniques are applied to increase the model or classifier classification ability and accuracy.

Before 1999, the most widely used techniques applied to microarray data analysis were correlation-based approaches. In some papers [9, 10], cluster analysis methods were used to group those genes that had similar patterns of gene expressions. This method provides very valuable information, but it suffers from noise affection. If the noise from the irrelative genes is not sufficiently reduced, it becomes difficult to correctly classify the samples.
In 1999, Golub et al. [8] proposed a neighborhood analysis approach to obtain informative genes. For each gene in the gene expression profile, a $P$ value is calculated using the formula: $P(g_i) = \frac{\mu_1(g_i) - \mu_2(g_i)}{\sigma_1(g_i) - \sigma_2(g_i)}$. The $P$ value gives the reference indicating how significantly the gene expression level differs between these two classes. A larger absolute value of $P$ signifies a more profound difference between the gene expression level of the two classes. Therefore, the genes are ranked by their $P$ values, and the top genes are chosen to perform the new class member prediction based on voting mechanisms. This method is a univariate approach as all samples are compared in a single dimension, on a gene-by-gene basis. It does not treat the genes collectively by analyzing the data in the N-dimensions. Furthermore, this method’s theory bases on that the relative genes are expressed uniformly.

In 2001 Leping Li et al. [3] presented a multivariate approach for performing sample classification by using gene subsets. A subset of genes consists of a multiple-gene dimension, and is used to classify the samples in this multi-gene dimension. It is obvious that the discriminative effect of certain genes are synergistic with the one-gene dimension based approach. It is believed that combinations of genes can be more discriminative for sample classification than individual domain genes. However, comparing all subsets of genes is not feasible in microarray datasets with thousands genes. It is in this context that genetic algorithms (GA) become useful. Genetic algorithms are stochastic search methodologies, and have been used to solve optimization problems involving high dimensional spaces. Therefore, it is quite suitable for subset gene selection. Genetic algorithms and the K-nearest neighbor method are combined to identify the many subsets
of genes in Li et al.’s approach. In this manner, the most frequently selected genes will consist of this multi-gene dimension. It has been shown that the selection frequency should relate with the relative predictive importance of genes for sample classification.

This thesis is based on the methods and ideas proposed by Li et al. The objective of this thesis is to identify the panel of discriminator genes (PDG) using a genetic algorithm. We consider the quality of gene expression, and add more strategies to increase our algorithm’s accuracy. Two sets of microarray profiles are used to test the stability and robustness of this algorithm. Dataset One is a new dataset upon which no data analysis has yet been done; this provides us a chance to explore the dataset without preconceptions. The general structure of this algorithm is as follows: first, a genetic algorithm is applied to find the near optimal chromosomes (NOC) which can classify all the samples correctly or can classify all samples above some criterion, e.g., 70% or 90% correct classification. The number of the NOC is 6000 for Dataset One, and 2000 for Dataset Two. After finding enough NOC, the top 50 genes are selected based on the frequency that they appear in the panel of NOC. The results demonstrate that the GA/KNN method can be an effective tool for classifying different subtypes of renal cell carcinoma (RCC) dataset, and acute myeloid leukemia (AML) - acute lymphoblastic leukemia (ALL) datasets.

The subject matter of the thesis is presented in the subsequent four chapters. Chapter 2 is devoted to an introduction to genetic algorithms in general, including the algorithm procedure and algorithm operations, and the implementation details used in our application. Chapter 3 presents the resulting data, as well as the methods and strategies used. Conclusions and discussion comprise the last chapter. Finally, the appendix
provides the critical portions of the source code, which includes the main program and the subroutines.
CHAPTER II
GENETIC ALGORITHMS

This chapter addresses the strategies and operations used in genetic algorithms and the implementation in our application. Genetic algorithms (GA) are a natural method for optimization, and a subset of evolutionary algorithms; these model biological processes to optimize highly complex cost functions. The method was developed by John Holland during the 1960s and 1970s, but in 1989, one of his students, David Goldberg, made this algorithm popular. In Goldberg’s dissertation, he solved a difficult problem involving the control of gas-pipeline transmission using a genetic algorithm.

2.1 Introduction to Genetic Algorithms

Genetic algorithms are searching processes based on the laws of natural selection and natural genetics. Generally, a simple GA will consist of three operations: Selection, Genetic Operation, and Replacement. A typical GA cycle is shown in Figure 2.1 [11]. First, the initial population is generated; this population comprises a group of chromosomes from which candidates can be selected for the solution of a problem. The initial population is generated randomly, and each of the chromosomes represents cascaded variables. After the initial generation is created, some selection schemes are used to evaluate the status of each chromosome. There are a number of methods, known as fitness techniques or cost evaluations, that are used to map the objective value to a
fitness value. Two commonly used techniques are windowing and parent selection. The fitness values of the all chromosomes are evaluated by calculating the objective function in a decoded form (phenotype). Next, a particular group of chromosomes (parents) is selected from the population to generate the offspring by the defined genetic operations. With a given parent selection scheme within the reproduction operator, more suitable chromosomes are selected from the population for further genetic enhancements. The fitness of the offspring is evaluated in a similar fashion to their parents. The chromosomes in the current population are then replaced by their offspring, based on a certain replacement strategy from two replacement strategies: generational-replacement and steady-state replacement.

Figure 2.1 Typical GA Cycle
This GA cycle is repeated until a desired termination criterion is reached. If all goes well throughout this process of simulated evolution, the best chromosomes in the final population will be highly evolved solutions to the problem.

GAs have several advantages: they provide a list of optimum parameters, not just a single solution; they can jump out of a local minimum; they can simultaneously search a wide sampling surface; they can handle large number of parameters; they are well suited for parallel computers, and are a good candidate for parallel applications. Together, these advantages produce stunning results when traditional optimization approaches fail.

2.2 Chromosome Representation

The encoding scheme is a key issue in any GA, as it can severely limit the window of the information observable from the system. The encoding scheme is the direct manipulation of coded variables that provides flexibility to solve different optimization problems using GAs. To enhance the performance of the algorithm, a chromosome representation that stores problem-specific information is desirable; it should be noted that each chromosome represents a trial solution to the problem setting. The chromosome is usually expressed as a string of variables, where each element is called a gene. The variable can be represented in binary, as a real number, or in other forms, and its range is usually defined by the problem specified.

Bit-string encoding is the most traditional approach used by GA researchers due to its simplicity and traceability. It is said that ordinary binary number representation of the parameter values may slow convergence of a genetic algorithm, so some people use gray codes to conquer this problem. The use of other encoding techniques, such as real
number representation, order-based representation (for bin-patching or graph coloring), embedded lists (for factory scheduling problems), variable element lists (for semiconductor layout), and even LISP S-expression, have been explored [4,5].

Integer-stream encoding method is used in our application. The parameter pool is a pool of integer numbers ranging from one to the maximum gene id. Each gene in a chromosome is chosen randomly from this parameter pool; due to the large amount of data chosen and from a statistical viewpoint, the duplication of genes in each chromosome is very unlikely, and thus we do not check for and eliminate them. The number of genes in each chromosome is set to 30 in our application. 30 genes from the gene pool are randomly selected and cascaded to form a chromosome. Figure 2.2 provides an example of a chromosome with ten genes.

```
220  26  1002  1803  75  1843  101  12  503  666
```

Figure 2.2 Chromosome Representation

In this example, 10 genes are randomly selected from the gene pool containing gene ids from one to 8,000, and a chromosome is created by cascading these 10 genes.

2.3 Fitness Function / Cost Function

The fitness function is the primary mechanism for evaluating the status of each chromosome and determining which parents should be selected to produce offspring. It takes a chromosome as input and produces a number of values as a measurement to the chromosome’s performance. A K-Nearest Neighbor (KNN) method is used as the fitness
function. It gives a percentage value depicting how the neighbors classify the sample to some type.

2.3.1 K-Nearest Neighbor Method

The nearest neighbor method is one of the most popular methods for classification and has been heavily investigated in the fields of statistics and pattern recognition. Even though more sophisticated alternative techniques have been developed, the mathematically simple non-parametric nearest neighbor methods still remain among the most successful for many classification problems. A more sophisticated version of this technique is k-nearest neighbor method. The idea behind the algorithm is quite straightforward. According to Massart et al. [6], in the KNN method “one selects the k nearest samples of sample x and classifies x in the group to which the majority of the samples belong.”

Normally, the KNN method is used to classify the unknown classes samples into classes based on the opinion of all of the k nearest neighbors. In our study, the KNN method is also used to measure how well the chromosome can classify the samples for which the class type information is already known. The measurement used in the thesis is a score between zero to one, and represents how well this chromosome can classify the samples to their classes or groups. For each gene combination, all K nearest neighbors must be found for each sample. If all of a sample’s K nearest neighbors belong to the same class, the ability of the chromosome to classify this sample is scored as one; otherwise, it is scored to zero. The scores of each sample are then added together, and
divided by the number of samples to yield the cost of the chromosome. A high score indicates that the chromosome can classify the classes of samples accurately.

2.3.2 KNN Performance and Drawbacks

The performance of the KNN method greatly depends on two factors: a suitable similarity function and an appropriate value for the parameter K. If the value of K is too great, large classes will overwhelm small ones; on the other hand, if K is too small, the advantages of the KNN algorithm will not exhibited. Setting K to a value between 3 and 5 is recommended [6]. The similarity functions will be discussed in the next section.

2.3.3 Distance Measures

In the prior section, we addressed the KNN method, without considering the methods used to define the similarity function; i.e., how to calculate the neighbors. The methods used to measure the samples’ closeness and similarity are of critical importance for KNN. There are a number of strategies to calculate distance between two samples in N-dimension space, including Euclidean distance, vector angle, and Pearson distance.

The Euclidean distance between two samples, sample x and y in N-dimensional space, is defined as follows $\sqrt{\sum_{i=1}^{N} (x_i - y_j)^2}$, where $i$ is the index of looping over the N dimensions. Vector angle between two samples x and y in N-dimensional space is defined as

$$ \cos \alpha = \frac{\sum_{i=1}^{N} x_i y_j}{\sqrt{\sum_{i=1}^{N} x_i} \sqrt{\sum_{i=1}^{N} y_i}} \quad (2.1) $$
Finally, Pearson distance between two samples $x$ and $y$ in $N$-dimensional space is defined in equation 2.2, where $\bar{x}$ and $\bar{y}$ are means.

$$\cos \alpha = \frac{\sum_{i=1}^{N} (x_i - \bar{x}) \sum_{i=1}^{N} (y_i - \bar{y})}{\sqrt{\sum_{i=1}^{N} (x_i - \bar{x})^2} \sqrt{\sum_{i=1}^{N} (y_i - \bar{y})^2}}$$

(2.2)

It is suggested that because the vector angle and Pearson distance show the trend of two samples, these two distances make more sense in a biological context than the Euclidean distance [11]. In our study, Euclidean distance and Pearson distance are both used in different runs for the two datasets, and the results are compared and analyzed in detail in Chapter 3.

2.4 Initial Population

Genetic algorithms start with a large amount of chromosomes known as initial population. The size of the initial population depends on the specification, and can be predicted from the constraints of the problem design. If the initial population is too large, the accuracy and performance of the genetic algorithm will suffer. Thus, we apply pre-selection based on the natural sectional (i.e., survival of the fittest) to discard large, high-cost chromosomes from the initial population.

2.4.1 Pre-selection

The initial generation is a two dimension matrix $N_{ipop} * N_{genes}$ with $N_{ipop}$ number of chromosome, and each chromosome has $N_{genes}$ of genes. First, we generate double-sized chromosomes, which have $2 * N_{ipop}$ chromosomes in the generation, and the chromosomes’ fitness values or cost values are calculated. Based on these cost values,
some pre-selection will be performed to increase the speed of the genetic algorithm and the chance to create optimal solutions. We rank the chromosomes from low cost to high cost, and then the top $N_{pop}$ chromosomes are chosen to create the initial generation. The initial population for this thesis is chosen to be 100 according to method outlined in Haupt and Haupt [4].

2.5 Genetic Operation

GAs generally have crossover and mutation operations; these are two ways to explore a cost surface.

2.5.1 Crossover

Crossover is a recombination operator that combines subparts of two parent chromosomes to produce offspring that contain parts of both parents’ genetic material. Many GA practitioners consider the crossover operator to be the determining factor that distinguishes the GA from all other optimization algorithms.

A number of variations on crossover operations have been proposed, with the simplest form being a single-point crossover. An example is shown in Figure 2.2.

Figure 2.3 Crossover Operation
The parents are selected based on the fitness selection scheme, and a crossover point is randomly selected. The portions of the two chromosomes beyond this point are exchanged, thus forming the offspring. To simplify the explanation, we need to give the position number, from the left. If we suppose the randomly selected position is five, the first child is generated in the following manner. First, we fill out the positions (zero to five) in the child from its father’s left side (positions zero to five). Then we fill out the child’s right side (from position six to nine) with the mother’s right side (position six to nine). The second child is created in the same manner, with the role of the father and mother switched.

Multipoint crossover can be used in genetic algorithms. Multipoint crossover is similar to single-point crossover, except that $m$ crossover positions are chosen at random with no duplication. Two-point crossover a common multipoint crossover strategy used in binary genetic algorithms. Two crossover points are selected randomly in the parents; they then swap the bits between the two crossover points. Another scheme involves three parents and two crossover points, and can produce a total of 18 offspring. For detail explanation, please refer to Haupt and Haupt [4].

2.5.1.1 Pairing

In the above section, we discuss the crossover operation. In this section, we will address how to choose the parent chromosomes to produce the offspring. Pairing chromosomes in a genetic algorithm is as varied as pairing in animal species. There are many methods to pair the chromosomes, such as pairing from top to bottom, random pairing, weighted random pairing (which is also called roulette wheel weighting, and
includes two techniques: rank weighting and cost weighting), and tournament selection. Our study uses the roulette wheel weighting method and cost weighting technique. It should also be noted that not all of the chromosomes can be a candidate for a parent.

2.5.1.1.1 Pairing from Top to Bottom

This is the simplest method for performing pairing, and a very good choice for the beginner. The idea and procedure are very straightforward: rank the chromosomes with their costing in this generation, and pair the chromosomes two at a time from the top to the bottom, such as Chromosome$_1$ pairing Chromosome$_2$, Chromosome$_3$ pairing Chromosome$_4$, and so on, until the desired number of parent pairs is generated.

2.5.1.1.2 Random Pairing

This selection is a uniform selection. It generates $N_{\text{good}}$ zero to one random numbers, multiples them by $N_{\text{good}}$, and then rounds the result to the nearest integer. These integer numbers correspond to the selected chromosomes’ number pairs.

2.5.1.1.3 Roulette Wheel

This selection process selects a new population with respect to the probability distribution based on the fitness values of the chromosomes [5]. A roulette wheel can be constructed as follows. First, the fitness value $f(c_i)$ for each chromosomes of the parents is calculated. Next, the total fitness value $F$, where $F = \sum_{i=1}^{\text{num \ population}} f(c_i)$, is calculated. Thirdly, we calculate the probability $p_i$ of a selection of a chromosome, where $p_i = f(c_i)/F$. Fourthly, we calculate a cumulative probability $q_i$ for each chromosome, where $q_i = \sum_{j=1}^{i} p_j$. Finally, a random number between zero and one is generated, starting from
the first chromosome of this generation chromosome list; the first chromosome with a cumulative probability greater than this random number is selected as a parent. After $N_{\text{parent}}$ chromosomes are selected, they will be grouped by twos to form parent pairs.

2.5.1.1.4 Tournament Selection

Tournament selection mimics mating competition in nature. It randomly picks a small subset of chromosomes from the mating pool, and the chromosome with the lowest cost in this subset becomes a parent. This procedure repeats until all needed parents are chosen.

2.5.2 Mutation

Mutation is an operator that introduces variations into the chromosomes. This variation can be global or local. The operation occurs occasionally but randomly alters the value of a string position. For the bit-stream encoding scheme, a randomly generated bit replaces a bit in the bit-string if a probability test is passed. Some GA practitioners use standard mutation to flip bits. When we implement the mutation operator, two issues should be answered: what types of mutation and what rate of mutation are to be used. There are diverse mutation methods used, such as multiple point mutation and single point mutation. Normally the mutation rate is kept low, because a high mutation rate may make the converging bouncing. The mutation rate used in our thesis is a random-based single point mutation, and the mutation rate is chosen to be 5% to 20%. We randomly choose the mutation point and chromosomes; however, the best chromosome is excluded from this process. We then randomly choose a gene id from the gene pool to replace the gene id in the mutation point of the chromosome.
2.6 Next Generation Creation

After crossover operation, the offspring are created, and mutation operations are performed. In this manner, the next generation is created and ready for the next iteration of the evolution. The strategy used in this study is based on a natural selection that can keep the nature of the parents, while also introducing some variation into the next generation.

Another approach to creating the next generation is to use thresholding. In the method, all chromosomes with lower costs than the threshold survive, but choosing the threshold may be difficult: it must allow enough parent chromosomes survive, while also permitting some variation among the offspring entering the next generation. After the threshold is chosen, it will be a fixed number throughout the algorithm execution.

2.7 GA Flowchart

A flowchart of a GA using KNN method flowchart is shown in Figure 2.4. First, the parameters are loaded. Then, the initial generation is created. The initial generation includes 200 chromosomes; each chromosome has 30 genes, which are picked randomly from the gene pool. After pre-selection, the top 100 chromosomes are selected, and the initial generation is created. This pre-selection is performed to increase the possibility of finding the optimal chromosomes. Next, the program goes through crossover and mutation operations. If optimal chromosomes are found, they are recorded and put into a panel of discriminated gene pool (PDG); crossover and mutation operations will be performed for five more loops before beginning a new iteration with a new initial
generation. If they are not found, we create the next generation and perform crossover and mutation operations until the maximum number of iteration is met. The above procedure continues until the specified number of optimal chromosomes are produced. The criteria to select optimal chromosomes are set from 75% to 100%, which indicates the accuracy with which the chromosome can classify all the samples.
Parameter setting, Vector data loading

# of optimal chromosomes is enough?

Yes

End

No

Create an initial generation

# of loops > Max_Loop

Yes

Crossover operation

Mutation operation

Create next generation

Any chromosomes good enough?

No

Yes

Create next generation

Loop five more times

End

Figure 2.4 GA Algorithm Flowchart
CHAPTER III
DATASET PROCESSING AND RESULTS ANALYSIS

This chapter presents our experimental results obtained from running the Genetic-KNN algorithm. Two different characteristic datasets are used, and many experiments are conducted to verify the algorithm's stability and robustness. The two datasets used have different characteristics: one has a very small set of samples with two types of classes and a larger number of genes, while the other is more complex, with more samples in each of two types of classes and a smaller number of genes.

The next section describes the datasets, and then the program procedure is explained in detail. Finally, the data results and data analysis are presented, as well as some strategies to increase the algorithm’s stability.

3.1 Datasets

In this thesis, two microarray data sets are used. One set of microarray gene expression profiles was obtained from Cleveland Clinic, and we refer to it as Dataset One in this thesis. It contains nine samples: six clear cell tumors and three papillary tumors, with over 18,000 genes expression in each sample. The other dataset is aml_all gene expression profiles, downloaded from the website [12], and we refer to it as Dataset Two. There are two classes of samples in Dataset Two, acute myeloid leukemia (AML) and acute lymphoblastic leukemia (ALL). There are 27 ALL and 11 AML with a total 38
samples in the training dataset, 21 ALL and 14 AML with a total 35 samples in the testing dataset.

The gene expression profiles contain gene quantative and quality expressions. Dataset One has quality scores. Dataset Two uses quality labels $A$, $M$ and $P$, where $A$ means absent, $M$ means marginal and $P$ means present. The quality labels and the quality score give a tool or a raw measure of how confidently we may trust the gene quantative expressions.

3.2 Program Flowchart (Procedure)

The program is implemented in MATLAB. It can be basically divided into four parts: data preprocessing, loading of initial values and settings, searching optimal chromosomes via our GA-KNN method, and selecting the top genes from optimal chromosome panel and classifying the testing samples.

3.2.1 Data Preprocessing Procedure

The preprocessing procedure is a very important part in classification and prediction. In this procedure, the noisy, missing or inconsistent data are been eliminated, which helps improve the quality of the data and consequently improves the classification and prediction results. There are some techniques that can be used in the data preprocessing procedure in data classification and prediction, such as data cleaning and data transformations. Data cleaning can be applied to remove noise and correct inconsistencies in the data. Data transformations are procedures that transfer or normalize the data to help mine useful information. Normalization is one of the most frequently
used data transformations. It can be applied to improve the classification algorithm involving distance measurement.

The data preprocessing performed in this program is an average threshold-based preselection procedure. The procedure is described as follows: for each gene in the original gene pool, the average gene quality score for this gene is calculated by adding together the gene quality scores among all the samples, and then dividing by the number of samples. If the value is equal or greater than the set threshold, we conclude that this gene qualitative expression value is reliable, so this gene is selected and put into the qualified gene pool. The above procedure loops through all the genes in the original pool. Before conducting experiments, each of the two datasets is fed to the preprocessing subroutine and qualified genes are selected. In Dataset One, the quality scores ranging from zero to one are provided in the gene expression profile. The average threshold is set at 0.5. After the preprocessing, the number of qualified genes or feature dimensions is reduced from 18,000 to 8,540. For Dataset Two, the quality scores are given as labels A, M, and P. We set A to be zero, M to be 0.5 and P to be one. The average threshold is set at 0.5. After this filtering procedure, the number of qualified genes or feature dimensions is reduced from 6817 to 1,683.

3.2.2 Parameters Setting, Genetic-KNN and Classification Procedure

The second step is to load the initial parameter setting, including such as the maximum number of genes in the gene pool, the gene number in each chromosome, the number of chromosomes in each generation, crossover rate, mutation rate, the optimal chromosome number, numbers of samples in each class, the $K$ value, the optimal
chromosome selection criteria, and the maximum number of iterations. After these values are set, we load the gene expressions and quality scores.

The next step is to go through the genetic-KNN algorithm procedure described in Chapter Two, using the training dataset to search for the near optimal chromosomes and filling out the near optimal chromosomes (NOC) panel. After the preset number of NOC is met, we proceed to the last step, in which the number of times that each gene appears in the NOC are counted and sorted, with the top 50 most frequently selected genes chosen and used to classify the testing dataset. The resulting classification accuracies are recorded. We take the quality score into consideration when selecting the top 50 genes, where the quality score is an indication of the reliability of a particular gene expression level. We use it to give more reliable data higher weight in the top 50 gene selection. Additionally, we select the top 50 gene without quality scores for experimental analysis purpose.

3.3 Data Results and Data Analysis

This section presents the experiments for these two datasets and the resulting data analysis. The experiments and data analysis using Dataset One are addressed first, followed by those for Dataset Two. Additionally, some strategies to increase the algorithm stability and robustness are presented for each dataset.

3.3.1 Dataset One

Due to Dataset One’s nature, namely having a small number of samples with too many genes, the criterion to select optimal chromosomes is set to one, NOC is set to 6000, and the $K$ is set to be two because the total number of the samples in clear cell RCC type
is three. As testing and training data are not separated in this dataset, the following experimental strategies are adopted. First, all samples are used as training data and testing data and the classification accuracies are recorded. Next, a bootstrap strategy is used to test the robustness of our algorithm, i.e., we take one sample out, and the rest act as training data. After the top 50 genes are selected, all samples are used as testing data and the classification accuracies are recorded. To increase the stability of this algorithm, the common genes from the six different experiments with the same setting are selected to do the all sample classification. For all the experimental strategies, two distance measurements, Euclidean and Pearson distances are used. With all samples used as both training and testing data, they can be correctly identified using the panel of discriminator genes (PDG) of 50 top genes from 6000 NOCs regardless of the type of distance or whether quality scores are used.

For the bootstrap strategy, we remove one papillary cell RCC sample at a time, use the remaining five papillary cells and three clear cell RCC samples to select the PDG, and then classify the removed sample using the selected PDG. Again, all six samples can be correctly identified regardless the type of distance or whether quality scores are used.

Table 3.1 Common Gene Percentages with All Samples for Dataset One

<table>
<thead>
<tr>
<th></th>
<th>Run One</th>
<th>Run Two</th>
<th>Run Three</th>
<th>Run Four</th>
<th>Run Five</th>
<th>Run Six</th>
</tr>
</thead>
<tbody>
<tr>
<td>Run One</td>
<td>1</td>
<td>0.87</td>
<td>0.83</td>
<td>0.8</td>
<td>0.83</td>
<td>0.83</td>
</tr>
<tr>
<td>Run Two</td>
<td>0.87</td>
<td>1</td>
<td>0.83</td>
<td>0.83</td>
<td>0.87</td>
<td>0.8</td>
</tr>
<tr>
<td>Run Three</td>
<td>0.83</td>
<td>0.83</td>
<td>1</td>
<td>0.83</td>
<td>0.87</td>
<td>0.83</td>
</tr>
<tr>
<td>Run Four</td>
<td>0.8</td>
<td>0.83</td>
<td>0.83</td>
<td>1</td>
<td>0.9</td>
<td>0.8</td>
</tr>
<tr>
<td>Run Five</td>
<td>0.83</td>
<td>0.87</td>
<td>0.87</td>
<td>0.9</td>
<td>1</td>
<td>0.8</td>
</tr>
<tr>
<td>Run Six</td>
<td>0.83</td>
<td>0.8</td>
<td>0.83</td>
<td>0.8</td>
<td>0.8</td>
<td>1</td>
</tr>
</tbody>
</table>

In table 3.1, the common gene percentages among six experiments are shown, with all samples acting as both the training and testing datasets. Table 3.2 shows the
common gene percentage among six different runs of same setting and with one sample removed at a time.

Table 3.2 Common Gene Percentages of Bootstrapping for Dataset One

<table>
<thead>
<tr>
<th></th>
<th>Run One</th>
<th>Run Two</th>
<th>Run Three</th>
<th>Run Four</th>
<th>Run Five</th>
<th>Run Six</th>
</tr>
</thead>
<tbody>
<tr>
<td>Run One</td>
<td>1</td>
<td>0.7</td>
<td>0.7</td>
<td>0.67</td>
<td>0.67</td>
<td>0.53</td>
</tr>
<tr>
<td>Run Two</td>
<td>0.7</td>
<td>1</td>
<td>0.8</td>
<td>0.8</td>
<td>0.8</td>
<td>0.73</td>
</tr>
<tr>
<td>Run Three</td>
<td>0.7</td>
<td>0.8</td>
<td>1</td>
<td>0.73</td>
<td>0.8</td>
<td>0.67</td>
</tr>
<tr>
<td>Run Four</td>
<td>0.67</td>
<td>0.8</td>
<td>0.73</td>
<td>1</td>
<td>0.77</td>
<td>0.63</td>
</tr>
<tr>
<td>Run Five</td>
<td>0.67</td>
<td>0.8</td>
<td>0.8</td>
<td>0.77</td>
<td>1</td>
<td>0.7</td>
</tr>
<tr>
<td>Run Six</td>
<td>0.53</td>
<td>0.73</td>
<td>0.67</td>
<td>0.63</td>
<td>0.7</td>
<td>1</td>
</tr>
</tbody>
</table>

Given the random nature of any GA-based method, we are concerned about the stability of the algorithm, i.e. how the final results change between different runs of the same code. To improve the consistency of the results and stability of the algorithm, we add an intersection step in the selection of top genes for the PDG. We run the code \( m \) times, select the set \( S_i, i=1,\ldots, m, \) of 50 top genes based on the frequency they appeared in the near optimal chromosomes, and then take the intersection of \( S_i, C=\cap S_i \) as the PDG.

Again, the all samples can be correctly identified regardless the type of distance or whether quality scores are used.

After the number of optimal chromosomes are selected and put in to the PDS, the gene id and the times selected are summarized. Figure 3.1 shows the gene id and the number of times it appears in PDG pool, with the horizontal axis showing the gene id, and the vertical axis depicting the number of times the genes appear in the PDG. Figure 3.2 shows the top 50 genes frequency diagram from the experiment. The horizontal axis represents the 50 genes that are selected most frequently, and the vertical axis represents the number of times that the genes are selected within the PDG pool.
Figure 3.1 Gene Ids and Number of Times Selected

Figure 3.2 Top 50 Genes Frequency for Dataset One
Table 3.3 shows the top 50 gene ids, the number of times the gene appears in the PDG and names from an experiment.

**Table 3.3 Top Gene Ids and Names from an Experiment for Dataset One**

<table>
<thead>
<tr>
<th>Gene Id</th>
<th>Frequency</th>
<th>Gene_Po</th>
<th>Gene_Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>7554</td>
<td>1095</td>
<td>16392</td>
<td>granzyme K (serine protease, granzyme 3; tryptase II)</td>
</tr>
<tr>
<td>3372</td>
<td>1077</td>
<td>7385</td>
<td>chromodomain protein, Y chromosome, 1</td>
</tr>
<tr>
<td>3614</td>
<td>690</td>
<td>7925</td>
<td>T cell receptor beta locus</td>
</tr>
<tr>
<td>2305</td>
<td>432</td>
<td>5014</td>
<td>regulator of G-protein signalling 5</td>
</tr>
<tr>
<td>2957</td>
<td>406</td>
<td>6452</td>
<td>interleukin 7 receptor</td>
</tr>
<tr>
<td>4727</td>
<td>342</td>
<td>10312</td>
<td>T cell receptor alpha locus</td>
</tr>
<tr>
<td>7625</td>
<td>277</td>
<td>16557</td>
<td>I-kappa-B-interacting Ras-like protein 2</td>
</tr>
<tr>
<td>4010</td>
<td>187</td>
<td>8779</td>
<td></td>
</tr>
<tr>
<td>4962</td>
<td>183</td>
<td>10785</td>
<td>regulator of G-protein signalling 5</td>
</tr>
<tr>
<td>6537</td>
<td>178</td>
<td>14154</td>
<td>DnaJ (Hsp40) homolog, subfamily A, member 3</td>
</tr>
<tr>
<td>8300</td>
<td>175</td>
<td>18062</td>
<td>collagen, type V, alpha 2</td>
</tr>
<tr>
<td>4517</td>
<td>123</td>
<td>9872</td>
<td>collagen, type IV, alpha 2</td>
</tr>
<tr>
<td>4259</td>
<td>120</td>
<td>9355</td>
<td>killer cell lectin-like receptor subfamily B, member 1</td>
</tr>
<tr>
<td>4749</td>
<td>109</td>
<td>10354</td>
<td>heparan sulfate (glucosamine) 3-O-sulfotransferase 1</td>
</tr>
<tr>
<td>4807</td>
<td>108</td>
<td>10468</td>
<td>secreted protein, acidic, cysteine-rich (osteonectin)</td>
</tr>
<tr>
<td>4257</td>
<td>107</td>
<td>9353</td>
<td>calcitonin receptor-like</td>
</tr>
<tr>
<td>906</td>
<td>103</td>
<td>1934</td>
<td></td>
</tr>
<tr>
<td>7148</td>
<td>101</td>
<td>15537</td>
<td>KIAA0982 protein</td>
</tr>
<tr>
<td>6599</td>
<td>101</td>
<td>14299</td>
<td>SPARC-like 1 (mast9, hevin)</td>
</tr>
<tr>
<td>6201</td>
<td>97</td>
<td>13438</td>
<td>cadherin 13, H-cadherin (heart)</td>
</tr>
<tr>
<td>1356</td>
<td>97</td>
<td>2914</td>
<td>Homo sapiens cDNA FLJ31066 fis, clone HSYRA2001153</td>
</tr>
<tr>
<td>3205</td>
<td>94</td>
<td>7004</td>
<td>FAT tumor suppressor homolog 1 (Drosophila)</td>
</tr>
<tr>
<td>6511</td>
<td>90</td>
<td>14098</td>
<td>KIAA0620 protein</td>
</tr>
<tr>
<td>92</td>
<td>89</td>
<td>193</td>
<td>37 kDa leucine-rich repeat (LRR) protein</td>
</tr>
<tr>
<td>6791</td>
<td>88</td>
<td>14746</td>
<td>RGC32 protein</td>
</tr>
<tr>
<td>6171</td>
<td>86</td>
<td>13383</td>
<td>granzyme A (granzyme 1, cytotoxic T-lymphocyte-associated serine esterase 3)</td>
</tr>
<tr>
<td>5736</td>
<td>85</td>
<td>12454</td>
<td>kinesin family member 3C</td>
</tr>
<tr>
<td>6424</td>
<td>80</td>
<td>13902</td>
<td>inositol 1,4,5-triphosphate receptor, type 3</td>
</tr>
<tr>
<td>6189</td>
<td>77</td>
<td>13418</td>
<td>CD3D antigen, delta polypeptide (TIT3 complex)</td>
</tr>
<tr>
<td>5253</td>
<td>76</td>
<td>11428</td>
<td>a disintegrin and metalloproteinase domain 19 (meltrin beta)</td>
</tr>
<tr>
<td>6134</td>
<td>75</td>
<td>13283</td>
<td>Immune associated nucleotide 4 like 1 (mouse)</td>
</tr>
<tr>
<td>2624</td>
<td>75</td>
<td>5698</td>
<td>secreted protein, acidic, cysteine-rich (osteonectin)</td>
</tr>
<tr>
<td>4333</td>
<td>74</td>
<td>9488</td>
<td>protein phosphatase 1, regulatory (inhibitor) subunit 3C</td>
</tr>
<tr>
<td>3514</td>
<td>72</td>
<td>7695</td>
<td>Finkel-Biskis-Reilly murine sarcoma virus (FBR-MuSV) ubiquitously expressed (fox derived); ribosoma</td>
</tr>
<tr>
<td>2574</td>
<td>72</td>
<td>5599</td>
<td></td>
</tr>
</tbody>
</table>
Table 3.3 Top Gene Ids and Names from an Experiment for Dataset One (Continuing)

<table>
<thead>
<tr>
<th>Gene Id</th>
<th>Rank</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>3417</td>
<td>67</td>
<td>EGF-containing fibulin-like extracellular matrix protein 1</td>
</tr>
<tr>
<td>6252</td>
<td>66</td>
<td>endothelin receptor type A</td>
</tr>
<tr>
<td>5999</td>
<td>64</td>
<td>Homo sapiens cDNA: FLJ22356 fis, clone HRC06345</td>
</tr>
<tr>
<td>3134</td>
<td>64</td>
<td>transcription factor 4</td>
</tr>
<tr>
<td>5272</td>
<td>63</td>
<td>EST</td>
</tr>
<tr>
<td>3426</td>
<td>63</td>
<td>KIAA0779 protein</td>
</tr>
<tr>
<td>2302</td>
<td>63</td>
<td>CGI-105 protein</td>
</tr>
<tr>
<td>5413</td>
<td>61</td>
<td></td>
</tr>
<tr>
<td>1770</td>
<td>60</td>
<td>interleukin 16 (lymphocyte chemoattractant factor)</td>
</tr>
<tr>
<td>5616</td>
<td>59</td>
<td>AE binding protein 1</td>
</tr>
<tr>
<td>7345</td>
<td>58</td>
<td>Gelsolin (amyloidosis, Finnish type)</td>
</tr>
<tr>
<td>6807</td>
<td>58</td>
<td>SPARC-like 1 (mast9, hevin)</td>
</tr>
<tr>
<td>6011</td>
<td>58</td>
<td>heparan sulfate proteoglycan 2 (perlecan)</td>
</tr>
<tr>
<td>5694</td>
<td>58</td>
<td>calpastatin</td>
</tr>
<tr>
<td>5121</td>
<td>58</td>
<td>tumor endothelial marker 5 precursor</td>
</tr>
</tbody>
</table>

We observed significant differences between the top genes selected using Euclidean and Pearson distances. In various test cases, only 45% - 50% of the top 50 genes from 6000 NOCs using the two distances are shared. Further, the final PDGs generated by the Euclidean and Pearson distances are not equivalent. For example, the PDG generated using the Euclidean distance can classify all nine samples correctly using the Pearson distance, while the PDG generated by the Pearson distance can only identify seven of the nine samples correctly using the Euclidean distance.

3.3.2 Dataset Two

Dataset Two has separate training and testing data, but the number of samples in each of the two classes is much greater than in Dataset One, and the number of genes after the data preprocessing is smaller as well. The criterion to select optimal chromosomes is set to 0.75, NOC is set to 2000 and $K$ is set to three. The following two experimental strategies are implemented. Training data are fed into the GA-KNN
algorithm; after finding the preset number of NOC, the most frequently selected genes are identified and used to classify the testing data. Six runs with the same settings are conducted, and the classification accuracies are recorded.

Table 3.4 Common Gene Percentages for Dataset Two

<table>
<thead>
<tr>
<th>Run</th>
<th>Run One</th>
<th>Run Two</th>
<th>Run Three</th>
<th>Run Four</th>
<th>Run Five</th>
<th>Run Six</th>
</tr>
</thead>
<tbody>
<tr>
<td>Run One</td>
<td>1</td>
<td>0.8</td>
<td>0.84</td>
<td>0.82</td>
<td>0.84</td>
<td>0.74</td>
</tr>
<tr>
<td>Run Two</td>
<td>0.8</td>
<td>1</td>
<td>0.76</td>
<td>0.8</td>
<td>0.82</td>
<td>0.78</td>
</tr>
<tr>
<td>Run Three</td>
<td>0.84</td>
<td>0.76</td>
<td>1</td>
<td>0.78</td>
<td>0.78</td>
<td>0.72</td>
</tr>
<tr>
<td>Run Four</td>
<td>0.82</td>
<td>0.8</td>
<td>0.78</td>
<td>1</td>
<td>0.82</td>
<td>0.78</td>
</tr>
<tr>
<td>Run Five</td>
<td>0.84</td>
<td>0.82</td>
<td>0.78</td>
<td>0.82</td>
<td>1</td>
<td>0.78</td>
</tr>
<tr>
<td>Run Six</td>
<td>0.74</td>
<td>0.78</td>
<td>0.72</td>
<td>0.78</td>
<td>0.78</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 3.4 shows the common genes percentage among these six runs from each top 50 genes. The accuracies of the classifications are recoded in Table 3.5. The testing data in Dataset Two has 21 samples for one class, Class One, and 14 samples for another class, Class Two. For all these experiments, the KNN value is set to 3, so the numbers zero to three shows how many K nearest neighbors of the sample classify it to a type. For example, in Run One, zero of the three nearest neighbors of sample number 18 evaluate it to Class One. In Run Three, its two nearest neighbors evaluate it to Class One. Accuracies presented in the Table 3.5 are ranged from 0 to 1, calculated as follows: the number of correctly classified samples is divided by the number of total samples.
Table 3.5 Top 50 Genes Classify Testing Dataset for Dataset Two

<table>
<thead>
<tr>
<th>Class</th>
<th>SampleId</th>
<th>Correct Classification Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class One</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(ALL)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>9</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>10</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>11</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>12</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>13</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>14</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>15</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>16</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>17</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>18</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>19</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>20</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>21</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Class Two</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(AML)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>9</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>10</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>13</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>14</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>
Subsequently, a bootstrap strategy is used to test the robustness of our algorithm: we take one sample out of the training data, use the rest of the dataset to act as training data, and after the top 50 genes are selected, they are used to classify the whole training dataset. A total of six runs are conducted and the classification accuracies are recorded in Table 3.6. Table 3.7 shows common genes and frequency information collected from the experiment.

Table 3.6 Bootstrap Tests for Dataset Two

<table>
<thead>
<tr>
<th>Experiments</th>
<th>Run 1</th>
<th>Run 2</th>
<th>Run 3</th>
<th>Run 4</th>
<th>Run 5</th>
<th>Run 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accuracy</td>
<td>0.974</td>
<td>1</td>
<td>0.947</td>
<td>0.974</td>
<td>0.974</td>
<td>0.947</td>
</tr>
<tr>
<td>Class</td>
<td>Sampled</td>
<td>Correct Classification Details</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Class One (ALL)</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>14</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>17</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>18</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>19</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>21</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>22</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>23</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>24</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>25</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>26</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>
Table 3.6 Bootstrap Tests for Dataset Two (Continuing)

<table>
<thead>
<tr>
<th>Class Two (AML)</th>
<th>27</th>
<th>3</th>
<th>3</th>
<th>3</th>
<th>3</th>
<th>3</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>9</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>10</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>11</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

The values of zeros to three, as well as the accuracy values, have the same meanings as in Table 3.5. The differences are the number of the samples in two classes.

Figure 3.3 Top 50 Genes Frequency for Dataset Two
### 3.7 Common Genes and Frequency Information Collected from Experiment

<table>
<thead>
<tr>
<th>Gene ID</th>
<th>Frequency</th>
<th>GenID_old</th>
<th># of A</th>
<th># of M</th>
<th># of P</th>
<th>Gene name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1103</td>
<td>383</td>
<td>4052</td>
<td>1103</td>
<td>0</td>
<td>1</td>
<td>72</td>
</tr>
<tr>
<td>729</td>
<td>364</td>
<td>2642</td>
<td>729</td>
<td>27</td>
<td>4</td>
<td>42</td>
</tr>
<tr>
<td>888</td>
<td>363</td>
<td>3258</td>
<td>888</td>
<td>11</td>
<td>3</td>
<td>59</td>
</tr>
<tr>
<td>1101</td>
<td>206</td>
<td>4050</td>
<td>1101</td>
<td>32</td>
<td>4</td>
<td>37</td>
</tr>
<tr>
<td>1512</td>
<td>199</td>
<td>5552</td>
<td>1512</td>
<td>3</td>
<td>0</td>
<td>70</td>
</tr>
<tr>
<td>1569</td>
<td>174</td>
<td>5772</td>
<td>1569</td>
<td>1</td>
<td>0</td>
<td>72</td>
</tr>
<tr>
<td>491</td>
<td>167</td>
<td>1779</td>
<td>491</td>
<td>34</td>
<td>2</td>
<td>37</td>
</tr>
<tr>
<td>1145</td>
<td>159</td>
<td>4211</td>
<td>1145</td>
<td>0</td>
<td>0</td>
<td>73</td>
</tr>
<tr>
<td>660</td>
<td>155</td>
<td>2354</td>
<td>660</td>
<td>25</td>
<td>0</td>
<td>48</td>
</tr>
<tr>
<td>367</td>
<td>154</td>
<td>1249</td>
<td>367</td>
<td>1</td>
<td>0</td>
<td>72</td>
</tr>
<tr>
<td>1367</td>
<td>150</td>
<td>4936</td>
<td>1367</td>
<td>0</td>
<td>0</td>
<td>73</td>
</tr>
<tr>
<td>1502</td>
<td>143</td>
<td>5501</td>
<td>1502</td>
<td>0</td>
<td>0</td>
<td>73</td>
</tr>
<tr>
<td>1178</td>
<td>129</td>
<td>4328</td>
<td>1178</td>
<td>0</td>
<td>0</td>
<td>73</td>
</tr>
<tr>
<td>550</td>
<td>128</td>
<td>1962</td>
<td>550</td>
<td>0</td>
<td>2</td>
<td>71</td>
</tr>
<tr>
<td>594</td>
<td>126</td>
<td>2111</td>
<td>594</td>
<td>9</td>
<td>2</td>
<td>62</td>
</tr>
<tr>
<td>159</td>
<td>124</td>
<td>464</td>
<td>159</td>
<td>0</td>
<td>0</td>
<td>73</td>
</tr>
<tr>
<td>1805</td>
<td>117</td>
<td>6803</td>
<td>1805</td>
<td>1</td>
<td>2</td>
<td>70</td>
</tr>
<tr>
<td>471</td>
<td>113</td>
<td>1704</td>
<td>471</td>
<td>0</td>
<td>0</td>
<td>73</td>
</tr>
<tr>
<td>1141</td>
<td>110</td>
<td>4196</td>
<td>1141</td>
<td>1</td>
<td>0</td>
<td>72</td>
</tr>
<tr>
<td>1669</td>
<td>105</td>
<td>6236</td>
<td>1669</td>
<td>1</td>
<td>0</td>
<td>72</td>
</tr>
<tr>
<td>1804</td>
<td>102</td>
<td>6797</td>
<td>1804</td>
<td>6</td>
<td>0</td>
<td>67</td>
</tr>
<tr>
<td>1740</td>
<td>97</td>
<td>6539</td>
<td>1740</td>
<td>15</td>
<td>8</td>
<td>50</td>
</tr>
<tr>
<td>269</td>
<td>96</td>
<td>804</td>
<td>269</td>
<td>8</td>
<td>0</td>
<td>65</td>
</tr>
<tr>
<td>1627</td>
<td>94</td>
<td>6041</td>
<td>1627</td>
<td>10</td>
<td>2</td>
<td>61</td>
</tr>
<tr>
<td>1657</td>
<td>91</td>
<td>6185</td>
<td>1657</td>
<td>15</td>
<td>1</td>
<td>57</td>
</tr>
<tr>
<td>179</td>
<td>90</td>
<td>532</td>
<td>179</td>
<td>0</td>
<td>0</td>
<td>73</td>
</tr>
<tr>
<td>1218</td>
<td>76</td>
<td>4461</td>
<td>1218</td>
<td>3</td>
<td>1</td>
<td>69</td>
</tr>
<tr>
<td>534</td>
<td>74</td>
<td>1928</td>
<td>534</td>
<td>1</td>
<td>0</td>
<td>72</td>
</tr>
<tr>
<td>342</td>
<td>71</td>
<td>1120</td>
<td>342</td>
<td>16</td>
<td>2</td>
<td>55</td>
</tr>
</tbody>
</table>

Figure 3.3 shows the top genes frequency from an experiment for Dataset Two.

Table 3.8 shows the gene ids, frequency information from an experiment for Dataset Two.

In the table the number of A, M, and P for each gene is recorded, as well as the gene name. In Figure 3.4, the genes frequency diagram is shown.
Table 3.8 Top Gene Ids and Names from an Experiment for Dataset Two

<table>
<thead>
<tr>
<th>Gene Id</th>
<th>Frequency</th>
<th>GO_#</th>
<th>GN_#</th>
<th>#_A</th>
<th>#_M</th>
<th>#_P</th>
<th>Gene_Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1103</td>
<td>398</td>
<td>4052</td>
<td>1103</td>
<td>0</td>
<td>1</td>
<td>72</td>
<td>X04085_rna1_at</td>
</tr>
<tr>
<td>729</td>
<td>336</td>
<td>2642</td>
<td>729</td>
<td>27</td>
<td>4</td>
<td>42</td>
<td>U05259_rna1_at</td>
</tr>
<tr>
<td>888</td>
<td>318</td>
<td>3258</td>
<td>888</td>
<td>11</td>
<td>3</td>
<td>59</td>
<td>U46751_at</td>
</tr>
<tr>
<td>1512</td>
<td>209</td>
<td>5552</td>
<td>1512</td>
<td>3</td>
<td>0</td>
<td>70</td>
<td>L06797_s_at</td>
</tr>
<tr>
<td>1569</td>
<td>208</td>
<td>5772</td>
<td>1569</td>
<td>1</td>
<td>0</td>
<td>72</td>
<td>U22376_cds2_s_at</td>
</tr>
<tr>
<td>1101</td>
<td>192</td>
<td>4050</td>
<td>1101</td>
<td>32</td>
<td>4</td>
<td>37</td>
<td>X03934_at</td>
</tr>
<tr>
<td>1145</td>
<td>173</td>
<td>4211</td>
<td>1145</td>
<td>0</td>
<td>0</td>
<td>73</td>
<td>X51521_at</td>
</tr>
<tr>
<td>491</td>
<td>173</td>
<td>1779</td>
<td>491</td>
<td>34</td>
<td>2</td>
<td>37</td>
<td>M19507_at</td>
</tr>
<tr>
<td>1178</td>
<td>155</td>
<td>4328</td>
<td>1178</td>
<td>0</td>
<td>0</td>
<td>73</td>
<td>X59417_at</td>
</tr>
<tr>
<td>1367</td>
<td>151</td>
<td>4936</td>
<td>1367</td>
<td>0</td>
<td>0</td>
<td>73</td>
<td>Y09433_at</td>
</tr>
<tr>
<td>367</td>
<td>150</td>
<td>1249</td>
<td>367</td>
<td>1</td>
<td>0</td>
<td>72</td>
<td>L08246_at</td>
</tr>
<tr>
<td>159</td>
<td>143</td>
<td>464</td>
<td>159</td>
<td>0</td>
<td>0</td>
<td>73</td>
<td>D50310_at</td>
</tr>
<tr>
<td>660</td>
<td>141</td>
<td>2354</td>
<td>660</td>
<td>25</td>
<td>0</td>
<td>48</td>
<td>M92287_at</td>
</tr>
<tr>
<td>550</td>
<td>135</td>
<td>1962</td>
<td>550</td>
<td>2</td>
<td>0</td>
<td>71</td>
<td>M33680_at</td>
</tr>
<tr>
<td>1502</td>
<td>131</td>
<td>5501</td>
<td>1502</td>
<td>0</td>
<td>0</td>
<td>73</td>
<td>Z15115_at</td>
</tr>
<tr>
<td>471</td>
<td>121</td>
<td>1704</td>
<td>471</td>
<td>0</td>
<td>0</td>
<td>73</td>
<td>M13792_at</td>
</tr>
<tr>
<td>594</td>
<td>114</td>
<td>2111</td>
<td>594</td>
<td>9</td>
<td>2</td>
<td>62</td>
<td>M62762_at</td>
</tr>
<tr>
<td>1805</td>
<td>112</td>
<td>6803</td>
<td>1805</td>
<td>1</td>
<td>2</td>
<td>70</td>
<td>M19045_f_at</td>
</tr>
<tr>
<td>269</td>
<td>107</td>
<td>804</td>
<td>269</td>
<td>8</td>
<td>0</td>
<td>65</td>
<td>HG1612-HT1612_at</td>
</tr>
<tr>
<td>867</td>
<td>106</td>
<td>3183</td>
<td>867</td>
<td>1</td>
<td>0</td>
<td>72</td>
<td>U41635_at</td>
</tr>
<tr>
<td>1657</td>
<td>105</td>
<td>6185</td>
<td>1657</td>
<td>15</td>
<td>1</td>
<td>57</td>
<td>X64072_s_at</td>
</tr>
<tr>
<td>1669</td>
<td>100</td>
<td>6236</td>
<td>1669</td>
<td>1</td>
<td>0</td>
<td>72</td>
<td>M23323_s_at</td>
</tr>
<tr>
<td>1804</td>
<td>92</td>
<td>6797</td>
<td>1804</td>
<td>6</td>
<td>0</td>
<td>67</td>
<td>J03801_f_at</td>
</tr>
<tr>
<td>1141</td>
<td>92</td>
<td>4196</td>
<td>1141</td>
<td>1</td>
<td>0</td>
<td>72</td>
<td>X17042_at</td>
</tr>
<tr>
<td>1627</td>
<td>91</td>
<td>6041</td>
<td>1627</td>
<td>10</td>
<td>2</td>
<td>61</td>
<td>L09209_s_at</td>
</tr>
<tr>
<td>534</td>
<td>87</td>
<td>1928</td>
<td>534</td>
<td>1</td>
<td>0</td>
<td>72</td>
<td>M31303_rna1_at</td>
</tr>
<tr>
<td>1599</td>
<td>86</td>
<td>5939</td>
<td>1599</td>
<td>0</td>
<td>0</td>
<td>73</td>
<td>J02621_s_at</td>
</tr>
<tr>
<td>715</td>
<td>85</td>
<td>2597</td>
<td>715</td>
<td>0</td>
<td>0</td>
<td>73</td>
<td>U02493_at</td>
</tr>
<tr>
<td>1740</td>
<td>83</td>
<td>6539</td>
<td>1740</td>
<td>15</td>
<td>8</td>
<td>50</td>
<td>X85116_rna1_s_at</td>
</tr>
<tr>
<td>1430</td>
<td>83</td>
<td>5191</td>
<td>1430</td>
<td>4</td>
<td>0</td>
<td>69</td>
<td>Z69881_at</td>
</tr>
<tr>
<td>481</td>
<td>82</td>
<td>1745</td>
<td>481</td>
<td>10</td>
<td>0</td>
<td>63</td>
<td>M16038_at</td>
</tr>
<tr>
<td>1149</td>
<td>81</td>
<td>4242</td>
<td>1149</td>
<td>0</td>
<td>0</td>
<td>73</td>
<td>X52851_rna1_at</td>
</tr>
<tr>
<td>232</td>
<td>79</td>
<td>668</td>
<td>232</td>
<td>0</td>
<td>2</td>
<td>71</td>
<td>D86967_at</td>
</tr>
<tr>
<td>342</td>
<td>76</td>
<td>1120</td>
<td>342</td>
<td>16</td>
<td>2</td>
<td>55</td>
<td>J04615_at</td>
</tr>
<tr>
<td>1218</td>
<td>74</td>
<td>4461</td>
<td>1218</td>
<td>3</td>
<td>1</td>
<td>69</td>
<td>X67698_at</td>
</tr>
<tr>
<td>1208</td>
<td>74</td>
<td>4407</td>
<td>1208</td>
<td>2</td>
<td>0</td>
<td>71</td>
<td>X64364_at</td>
</tr>
<tr>
<td>1082</td>
<td>74</td>
<td>3984</td>
<td>1082</td>
<td>0</td>
<td>0</td>
<td>73</td>
<td>U94855_at</td>
</tr>
<tr>
<td>408</td>
<td>73</td>
<td>1394</td>
<td>408</td>
<td>2</td>
<td>2</td>
<td>69</td>
<td>L20941_at</td>
</tr>
<tr>
<td>1788</td>
<td>72</td>
<td>6732</td>
<td>1788</td>
<td>1</td>
<td>0</td>
<td>72</td>
<td>X89399_s_at</td>
</tr>
<tr>
<td>1533</td>
<td>72</td>
<td>5642</td>
<td>1533</td>
<td>0</td>
<td>0</td>
<td>73</td>
<td>M19311_s_at</td>
</tr>
<tr>
<td>508</td>
<td>72</td>
<td>1829</td>
<td>508</td>
<td>31</td>
<td>0</td>
<td>42</td>
<td>M22960_at</td>
</tr>
<tr>
<td>179</td>
<td>72</td>
<td>532</td>
<td>179</td>
<td>0</td>
<td>0</td>
<td>73</td>
<td>D63874_at</td>
</tr>
<tr>
<td>1655</td>
<td>71</td>
<td>6181</td>
<td>1655</td>
<td>0</td>
<td>0</td>
<td>73</td>
<td>M14483_rna1_s_at</td>
</tr>
<tr>
<td>1672</td>
<td>70</td>
<td>6242</td>
<td>1672</td>
<td>0</td>
<td>0</td>
<td>73</td>
<td>M24485_s_at</td>
</tr>
</tbody>
</table>
Table 3.8 Top Gene Ids and Names from an Experiment for Dataset Two (Continuing)

<table>
<thead>
<tr>
<th>Gene Ids</th>
<th>Number of Times Selected</th>
</tr>
</thead>
<tbody>
<tr>
<td>704</td>
<td>63</td>
</tr>
<tr>
<td>1166</td>
<td>0</td>
</tr>
<tr>
<td>1531</td>
<td>0</td>
</tr>
<tr>
<td>933</td>
<td>0</td>
</tr>
<tr>
<td>468</td>
<td>0</td>
</tr>
<tr>
<td>1806</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 3.4 Gene Ids and Number of Times Selected for Dataset Two
CHAPTER IV
CONCLUSION AND DISCUSSION

We have developed an effective tool for classifying different subtypes of RCC and ALL-AML. We tested the classification accuracy of the panel of discriminator genes (PDG) to different samples; it shows that 100% accuracy can be achieved for Dataset One, and 88.6% to 91.4% (i.e., 31 to 32 out of 35 samples) can be correctly classified for Dataset Two. We tested the stability of the algorithm, and showed that across multiple runs, 67% to 90% of the top 50 genes are in common. We presented a method to reduce the sensitivity of the PDG to different samples. We exploited the effects of quality scores on the PDG; it shows a difference of about 10%. Furthermore, we analyzed the different distance metrics effects on the PDG for Dataset One.

From these procedures, we arrived at the conclusion that this genetic algorithm with the K-nearest neighbor method is an effective method to select a subset of discriminator genes based on microarray data, and an effective tool for classifying two types of gene classes. Furthermore, such a set of discrimination genes can be used to classify unknown samples.

Because of the limited scope of the present study, the following paths are recommended for future study: parallel implementation, which can dramatically decreases the execution time of experiments; in-depth analysis of the effect of different distances on the selection of PDGs; possible development of more suitable distance
metrics for microarray data analysis; testing of the method using more complex datasets, such as those with more than two classes and more samples in each class; exploring $K$ effects to the algorithm and classification; and optimizing the algorithms for the KNN method.
REFERENCES


    or ftp://ftp.broad.mit.edu/distribution/genepattern/datasets/all_aml/
APPENDICES
APPENDIX A

CRITICAL SOURCE CODE

all_amlProg_total.m

This function is the main function for testing all_aml dataset.
1. read in setting parameters stored in InputData_1863.txt
2. load training dataset, save to SamplesVectors(samples as rows, gene expression values as columns) and SamplesQuiltVector(same dimension)
3. after the genetic algorithm finishes, saves the necessary files, and resulting data.
4. input
   StartValue    EndValue
   ChromNum (# of chromosomes for each generation)
   GeneNum (# of genes for each Chrom)  FlagD1(-----)  CrossEable  CrossRate
   MutationEable  MutationRate
   NumR2 (# of optimal (best, meet the measurement) chromosomes)
   ClassNums (total samples number)
   Class1Num (# of samples in class one) Class2Num(# of samples in class two)
   KNum (# of nearest neighbors)
   Measure (the critical to select optimal chromosomes)
   LoopNum (the max # of loops for each generation evolution)
   TopGenNum (the # of top genes)
   partialFlag    (flag for considering quality of gene expressions)

clear
StartTime = cputime;

[Names,Values] = textread('InputData_1863.txt','%s  %f',18);
StartValue = Values(1);  EndValue = Values(2);         ChromoNum = Values(3);
GeneNum = Values(4);   FlagD1 = Values(5);             CrossEable = Values(6);
MutationEable = Values(7);  MutationRate = Values(8);
NumR2 = Values(10);     ClassNums = Values(11);     Class1Num = Values(12);
Class2Num = Values(13);  KNum = Values(14);         Measure = Values(15);
LoopNum = Values(16); TopGenNum = Values(17); partialFlag = Values(18);

%-------------------------------------------------------------------------------------------------------------------------------
% read samples information in from training or test data set.
%-------------------------------------------------------------------------------------------------------------------------------

a = load('TrainData_05.txt') ;
a = a';
SampleNums = size(a);

k = 1;
for t = 1:2:SampleNums(1)           % # of samples*2
    SamplesVectors(k,:) = a(t,:);  k = k+1;
    % assign the vector expression values, dimension(samples*genes)
end

k = 1;
for t = 2:2:SampleNums(1)           % # of samples*2
    SamplesQuiltVector(k,:) = a(t,:); k = k+1;
    % assign quality expression values, dimension(samples*genes)
end

size(SamplesVectors)

%-------------------------------------------------------------------------------------------------------------------------------
% assign the samples classes
%-------------------------------------------------------------------------------------------------------------------------------

ClassNumVector(1) = Values(12); ClassNumVector(2) = Values(13);
SampleClass = SampleType(ClassNumVector, ClassNums)

for t = 1:Class1Num + Class2Num
    TrainingSample(t) = t;
end

% open file to save the best results
FId = fopen('results_05_3000.txt', 'w');
ResultsNum = 0;                                             % # of NOC
looptimes = 0;
while (ResultsNum <= NumR2)                                   % to get NumR2 # of NOC
    looptimes = looptimes + 1;
    ResultsNum
%-------------------------------------------------------------------------------------------------------------------------------
% create double sized initial generation, then choose the best ChromoNum
% # of chromosomes to construct the initial generation
%-------------------------------------------------------------------------------------------------------------------------------

[GenIntD, Flag] =
    InitGenFunc(StartValue, EndValue, 2*ChromoNum, GeneNum, FlagD1);
for t=1:ChromoNum*2
    [FitnessValueD(t),SampleScores,TDouble(t,:)] = ...
        FitnessIntFunc(t,GenIntD,KNum,SamplesVectors,SampleClass,…
            TraingSample,SamplesQuiltVector,partialFlag) ;
end

[SortedFitnessValueD,SortedDIndex]= sort(1-FitnessValueD);
for t=1:ChromoNum
    GenInt(t,:)=GenIntD(SortedDIndex(t,:),);
end

loop_t=0;
BestFit = 0;
for t=1:ChromoNum
    [FitnessValue(t),SampleScores,T(t,:)] = ...
        FitnessIntFunc(t,GenInt,KNum,SamplesVectors,SampleClass,…
            TraingSample,SamplesQuiltVector,partialFlag) ;
end
[SortedFitnessValue,SortedIndex]= sort(1-FitnessValue);

flag=0;
while (loop_t <= LoopNum & flag < 5  )
    if(BestFit >= Measure) flag = flag+1;
    else                   loop_t=loop_t +1;
    end
    loop_t

    sum_fitnessvalue(loop_t) = sum(FitnessValue) ;
    GoodNum=ChromoNum - fix(CrossRate*ChromoNum);
    for t=1:GoodNum-1
        GenIntMutIn(t,:)=GenInt(SortedIndex(t+1),:);
        % no mutation for  the best chromosome ######
        FitnessValueMutIn(t) = FitnessValue(SortedIndex(t+1));
    end
    BestChromosome(1,:) =GenInt(SortedIndex(1),:);

%--------------------------------------------------------------------------
% genetic operators
%--------------------------------------------------------------------------
% crossover
    [CrossOutInt ,OffspringNum]= …
        CrossOverFunc(FitnessValue, CrossRate,GenInt,StartValue,EndValue);

    SizeCrossOutInt = size(CrossOutInt);
    for t=1:SizeCrossOutInt(1)
GenIntMutIn(t+GoodNum-1,:) = CrossOutInt(t,:);
end
for t=1:SizeCrossOutInt(1)
    [FitneesValueMutIn(t+GoodNum), SampleScoresCross, TCross(t,:)] = ... 
        FitnessIntFunc(t, CrossOutInt, KNum, SamplesVectors, SampleClass, ... 
            TraingSample, SamplesQuiltVector, partialFlag) ;
end

% mutation
    [GenMutationInt, MutationPosition] = MutationIntFunc(FitneesValueMutIn, ... 
        MutationRate, GeneNum, GenIntMutIn, StartValue, EndValue);

% next generation construction, bestone + GenMutationInt
GenNext(1:ChromoNum-1,:) = GenMutationInt(1:ChromoNum-1,:); 
GenNext(ChromoNum,:) = BestChromosome(1,:); 
for t=1:ChromoNum
    [FitnessValue(t), SampleScores, T(t,:)] = ... 
        FitnessIntFunc(t, GenNext, KNum, SamplesVectors, SampleClass, ... 
            TraingSample, SamplesQuiltVector, partialFlag) ;
end

[SortedFitnessValue, SortedIndex] = sort(1-FitnessValue);

GenInt = GenNext;

BestChrom(loop_t,:) = GenNext(SortedIndex(1),:); 
FitnessValueBest(loop_t) = FitnessValue(SortedIndex(1));

% for display
GenInt(SortedIndex(1),:);
BestFit = FitnessValue(SortedIndex(1))

end
RecordLoopTimes(looptimes) = loop_t;
%--------------------------------------------------------------------------
% save NOC chromosomes from each run results
%--------------------------------------------------------------------------
if(loop_t <= LoopNum)
    [BestNum, FitnessValueS, SortedIndexS, GenResultsInt] = ... 
        SaveResultsFunc(GenInt, ChromoNum, KNum, SamplesVectors,... 
            SampleClass, Fld, Measure, TraingSample, SamplesQuiltVector, partialFlag);
    GenTotalResultsInt(ResultsNum+1:ResultsNum+BestNum,:) = ... 
        GenResultsInt(1:BestNum,:);
    ResultsNum = ResultsNum + BestNum
end
clear GenIntD,GenIntMutIn,GenMutationInt,GenInt;
clear Fit*;
clear SampleScores;
clear T,TDouble ,TCross;
clear Sorted*;
clear sum_fitnessvalue;
clear BestChrom,CrossOutInt,OffspringNum,MutationPosition,Flag;
end             % while end for ResultsNum < NumR2

fclose(FId);
ExecutionTime = cputime - StartTime;

% the following is for the top genes selection purely based on the frequency
[GeneFrequency,GeneSelectedNum,GeneFrequencyN] = …
    GeneFrequencyFunc(GenTotalResultsInt,EndValue);

fid2=fopen('freqresult_05_3000.txt','w');
for t=1:GeneSelectedNum
    fprintf(fid2,'%6d %6d 
',GeneFreqency(t,2), GeneFreqency(t,1));
end
fclose(fid2);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% read in the test samples' vector values ( expression values and quality values),
% read in the class information
% set the proper TraingSample vector

[Names,Values] = textread('InputData_test_1863.txt','%s  %f',18 );
StartValue = Values(1); EndValue = Values(2); ChromoNum= Values(3);
GeneNum = Values(4);   FlagD1 = Values(5);    CrossEable = Values(6);
CrossRate = Values(7);   MutationEable = Values(8); MutationRate = Values(9);
NumR2 = Values(10);     ClassNums = Values(11);     Class1Num = Values(12);
Class2Num = Values(13);   KNum = Values(14);      Measure = Values(15);
LoopNum=Values(16);    TopGenNum = Values(17); partialFlag=Values(18);

a = load('TestData_05.txt') ;  a=a';   
SampleNums = size(a);

k=1;
for t=1:2:SampleNums(1)   % # of samples*2
    SamplesVectors(k,:) = a(t,:);      k=k+1;
end
k=1;
for t=2:2:SampleNums(1)   % # of samples*2
SamplesQuiltVector(k,:) = a(t,:); k=k+1;
end
size(SamplesVectors)

ClassNumVector(1)=Values(12); ClassNumVector(2)=Values(13);
SampleClass = SampleType(ClassNumVector,ClassNums)

for t=1:Class1Num+Class2Num
    TotalSample(t)=t;
end

[freqsort, freqindex]=sort(GeneFrequency(:,1));
sortlistlen = length(GeneFrequency);
topgenenumd =TopGenNum*2;
for t=0:topgenenumd-1
    SortedGeneFreqency(t+1,1)=GeneFrequency(freqindex(sortlistlen-t),1); % frequency
    SortedGeneFreqency(t+1,2)=GeneFrequency(freqindex(sortlistlen-t),2); % gene id
end

for t=0:TopGenNum-1
    GenSelectTest(1,t+1)=SortedGeneFrequency(t+1,2);
    [FitnessValueTest(1,t+1),SampleScoresTest(1,t+1),TSelectTest(t+1,:)] = ...
        TestFitnessFunc(1,GenSelectTest,KNum,SamplesVectors,SampleClass,TotalSample);
end

fpp=fopen('topgeneoriginal.txt','w');
fpn=fopen('NewName05.txt','r');
TopGeneFile_1(fpp,fpn,SortedGeneFreqency,FitnessValueTest,SampleScoresTest,…
    TSelectTest,TopGenNum);
fprintf(fpp,'execution time is %12.2f

',ExecutionTime);
fclose(fpp);fclose(fpn);

for t=1:TopGenNum;
    bar(tp,SortedGeneFreqency(1:TopGenNum,1));
    saveas(gcf,'freq_orig.fig'); % note, gcf can not be changed to other name ********##
    close(gcf);

    % write doubled # of TopGenNum samples genes (original, no freq weight) expressions,
    % qualities and other information into a file.
    fpe=fopen('topgenexpression.txt','w');
    dgenenum=TopGenNum*2;
    samplen= length(TotalSample);
    for t=1:dgenenum
        for tt=1:samplen
            TopGExpV(t,tt)=SamplesVectors(tt,SortedGeneFreqency(t,2))    ;
    end
end
TopGExpQ(t,tt)=SamplesQuiltVector(tt,SortedGeneFrequency(t,2));
fprintf(fpe,'%12.6f ',TopGExpV(t,tt));
fprintf(fpe, '%5.2f ',TopGExpQ(t,tt));
end
fprintf(fpe,'\n');
end
fclose(fpe);

% save the distances among the samples using not weighted top genes
% file names are ['Origsample' int2str(t) '.fig'], t is the sample name
tlen=length(TotalSample);
GeneTestOrig(1:TopGenNum,1)=SortedGeneFrequency(1:TopGenNum,2);
GeneTestOrig=GeneTestOrig';
for t=1:tlen
    for tt=1:tlen
        DistOrig(t,tt) = …
        Distance(SamplesVectors,TotalSample(t),TotalSample(tt),1,GeneTestOrig);
    end
end

for t=1:tlen
    polar(2*pi.*TotalSample/(tlen+1),DistOrig(t,:),'-or')
title(['Origsample' int2str(t) ' distance'],'Color','b'),
% the title won't show in the figure, if use menu,insert, it will show, then dispear ???
% saveas function, no title is added to the figure
    saveas(gcf,['Origsample' int2str(t) '.fig']);
close(gcf);
end

% chooses top genes partially based on frequency and the fitness score
[numTotalChromosomes,chromosomeLength] = size(GenTotalResultsInt);

% this is to record all the chosen chromosomes(NOC chromosomes) ' fitnessscores
for t=1:numTotalChromosomes
    [fitnessScores(1,t),templeSampleScores(1,t), tempTDouble(t,:)]= ...
    FitnessIntFunc(t,GenTotalResultsInt,KNum,SamplesVectors,SampleClass,…
    TotalSample,SamplesQuiltVector,partialFlag) ;
end

% take the gene's avg quality among the samples into account while moving
% top genes forward or backward on the top list
GeneFrencyResult = ...
    TopGeneSelectFunc(GenTotalResultsInt,EndValue,fitnessScores,…
    SamplesQuiltVector,TopGenNum);
% [i,1] the frequency of the gene, [i,2] the gene id

% calculate the fitness value of the gene combination, not use gene quality
for t=0:TopGenNum-1
    GenSelectTestWeight(1,t+1)=GeneFrencyResult(t+1,2);
    [FitnessValueTestWeight(1,t+1),SampleScoresTestWeight(1,t+1),
     TSelectTestWeight(t+1,:)] = ...
        TestFitnessFunc(1,GenSelectTestWeight,KNum,SamplesVectors,…
                    SampleClass,TotalSample);
end

% file 'topgeneweight.txt' recodes the weighted top genes
fppw=fopen('topgenweight.txt','w');
fpnw=fopen('NewName05.txt','r');
TopGeneFile_1(fppw,fpnw,GeneFrencyResult,FitnessValueTestWeight,…
    SampleScoresTestWeight,TSelectTestWeight,TopGenNum);
fprintf(fppw,'execution time is %12.2f

',ExecutionTime);
fclose(fppw); fclose(fpnw);

% save the distances among the samples using weighted top genes
% file names are ['sample' int2str(t) '.fig'], t is the sample name
% tlen=length(TotalSample);
GeneTest(1:TopGenNum,1)=GeneFrencyResult(1:TopGenNum,2);
GeneTest=GeneTest';
for t=1:tlen
    for tt=1:tlen  % gene position
        Dist(t,tt) = Distance(SamplesVectors,TotalSample(t),TotalSample(tt),1,GeneTest);
    end
end

CrossOverFunc.m

function CrossOverFunc

% This function do paring and then perform crossover operation

for t=0:TopGenNum-1
    GenSelectTestWeight(1,t+1)=GeneFrencyResult(t+1,2);
    [FitnessValueTestWeight(1,t+1),SampleScoresTestWeight(1,t+1),
     TSelectTestWeight(t+1,:)] = ...
        TestFitnessFunc(1,GenSelectTestWeight,KNum,SamplesVectors,…
                    SampleClass,TotalSample);
end

% file 'topgeneweight.txt' recodes the weighted top genes
fppw=fopen('topgenweight.txt','w');
fpnw=fopen('NewName05.txt','r');
TopGeneFile_1(fppw,fpnw,GeneFrencyResult,FitnessValueTestWeight,…
    SampleScoresTestWeight,TSelectTestWeight,TopGenNum);
fprintf(fppw,'execution time is %12.2f

',ExecutionTime);
fclose(fppw); fclose(fpnw);

% save the distances among the samples using weighted top genes
% file names are ['sample' int2str(t) '.fig'], t is the sample name
% tlen=length(TotalSample);
GeneTest(1:TopGenNum,1)=GeneFrencyResult(1:TopGenNum,2);
GeneTest=GeneTest';
for t=1:tlen
    for tt=1:tlen  % gene position
        Dist(t,tt) = Distance(SamplesVectors,TotalSample(t),TotalSample(tt),1,GeneTest);
    end
end

for t=1:tlen
    polar(2*pi.*TotalSample/(tlen+1),Dist(t,:),'-or')
    hold on
    title(['sample' int2str(t) ' distance'],'Color','b'),
    saveas(gcf,['sample' int2str(t) '.fig']);
    close(gcf);
end
function [CrossOutInt, OffspringNum] = CrossOverFunc(FitnessValue, CrossRate, GenInt, StartValue, EndValue)

size_chrom = size(GenInt);
SelectNum = fix(CrossRate * size_chrom(1));
GenSelectOutInt = RouletteWheelSelectFunc(GenInt, FitnessValue, SelectNum);

size_parent = size(GenSelectOutInt);
even_size = mod(size_parent(1), 2);
if (even_size==0)
    for t=1:2:size_parent(1)
        CrossOutInt(t:t+1,:) = CrossOperation(GenSelectOutInt(t,:), GenSelectOutInt(t+1,:), StartValue, EndValue);
    end
else % if the # of parents is not even, truncate the last one
    for t=1:2:size_parent(1)-1
        CrossOutInt(t:t+1,:) = CrossOperation(GenSelectOutInt(t,:), GenSelectOutInt(t+1,:), StartValue, EndValue);
    end
end

OffspringNum = fix(size_parent(1));

 tướng                                                                      *******Distance.m *******

function Dist_Co = Distance(SamplesVectors, Sample1, Sample2, GenIntIndex, GenInt)

numGene = size(GenInt);

mean1=0; mean2=0;
for t=1:numGene(2)
    GenPosition = GenInt(GenIntIndex,t);
    mean1= mean1 + SamplesVectors(Sample1,GenPosition);
    mean2= mean2 + SamplesVectors(Sample2,GenPosition);
end

mean1= mean1 /numGene(2) ;
mean2= mean2 / numGene(2);
nominator1=0; nominator1=0; nominator2=0;
for t=1:numGene(2)
    GenPosition = GenInt(GenIntIndex,t);
    nominator1= nominator1+ (SamplesVectors(Sample1,GenPosition)- ... 
    mean1)*(SamplesVectors(Sample2,GenPosition)-mean2) ;
    nominator1 = nominator1+ (SamplesVectors(Sample1,GenPosition)-mean1)^2 ;
nominator2 = nominator2 + (SamplesVectors(Sample2,GenPosition)-mean2)^2;
end
if ( (nominator1==0)||(nominator2==0) ) % the divided by zero need to be checked
    Dist_Co=0;
else
    Dist_Co = 1- numerator1/(sqrt(nominator1) *sqrt(nominator2));
end
if(length(GenInt)==1)
    GenPosition = GenInt(1,1);
    Dist_Co=abs(SamplesVectors(Sample1,GenPosition)- …
    SamplesVectors(Sample2,GenPosition) )
End

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Distance_angle.m
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
def Distance_angle = …
function Dist_angle = …
    Distance_angle(SamplesVectors,Sample1,Sample2,GenIntIndex,GenInt)
numGene = size(GenInt)
Dist=0; numerator1=0; nominator1=0;nominator2=0;
for t=1:numGene(2)
    GenPosition = GenInt(GenIntIndex,t);
    numerator1= numerator1 + SamplesVectors(Sample1,GenPosition)…
    *SamplesVectors(Sample2,GenPosition) ;
    nominator1 = nominator1 + (SamplesVectors(Sample1,GenPosition) )^2 ;
    nominator2 = nominator2 + (SamplesVectors(Sample2,GenPosition) )^2 ;
end
Dist_angle = 1- numerator1/ ( (sqrt(nominator1)) *( sqrt(nominator2))

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
FitnessIntFunc.m
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
def FitnessIntFunc = …
function FitnessIntFunc = …
    FitnessIntFunc = …
% This is the fitness function, to evaluate a chromosome's classification ability
% 1. find the samples Knn
% 2. check if they all agree on the class types
% 3. assign the score R2 to this chromosome(GenIntIndex)
%
% Input :
% GenIntIndex : the index in the GenInt Array( chromosomes array)
% GenInt : chromosomes array
% Knum : the # of neighbors

50
% SamplesVectors : samples values, used to calculate the distance between samples.
% SampleClass : the # of class types, for us, there are two
% Output :
% FitnessValue : the fitness for the chromosome (how good this chromosome to
% classify the samples to classes)
% = sum(Score) / # of the samples
% SampleScores : sample score is the # of the samples to be classified by the Chro
% T : the # of agrees
%
% may 12, 2004, the fitness function need to take the sample gene quality into
% consideration, i.e., add weight to each sample's chosen genes
% one parameter (SamplesQuiltVectors) is added

function [FitnessValue, SampleScores, T] = ...
    FitnessIntFunc(GenIntIndex, GenInt, KNum, SamplesVectors, ...
                   SampleClass, TraingSample, SamplesQuiltVector, partialFlag)

[SortDistNeighbors, NeighborIndexKNN] = …
    KNN(SamplesVectors, GenIntIndex, GenInt, KNum, TraingSample);

TraingLength = length(TraingSample);
for t = 1:TraingLength
    T(t) = AgreeFunc(NeighborIndexKNN, t, KNum, SampleClass, TraingSample);
    % that means, the chromosome's partial classification need to be considered
    if(partialFlag == 1)
        Score(t) = T(t) / KNum;
        if(T(t) == KNum) Score(t) = 1;
        else Score(t) = 0;
        end
    else
        Score(t) = 0;
    end

    PQuality = AverageQualityFunc(GenInt(GenIntIndex,:),...
                                   SamplesQuiltVector, TraingSample(t));
    Score(t) = Score(t) * PQuality;
end

FitnessValue = sum(Score) / TraingLength;
SampleScores = sum(Score);

/*GeneFrequencyFunc.m

GeneFrequencyFunc.m

function [GeneFrequency, GeneSelectedNum, GeneFrequencyN] = …

51
```matlab
GeneFrequencyFunc(GenInt,RangNum)

t=1:1:RangNum ;
GeneFrequencyN = hist(GenInt,t');

for tt=1:RangNum
    GeneFrequencyTemp(tt,1)= sum(GeneFrequencyN(tt,:));
end

k=0;
for kk=1:RangNum
    if (GeneFrequencyTemp(kk)~=0)
        k=k+1;
        GeneFrequency(k,1)=GeneFrequencyTemp(kk);
        GeneFrequency(k,2)= kk;
    end
end
bar(GeneFrequency(:,2),GeneFrequency(:,1) )
GeneSelectedNum = k;

saveas(gcf,'Frequencyhistgram.fig'); close(gcf);
```

---

```matlab
InitGenFunc.m

function [GenInt,Flag] = InitGenFunc(StartValue,EndValue,ChromoNum,GeneNum,FlagD1)

Flag=zeros(1,ChromoNum); t=1;
while (t<=ChromoNum)
    flagT=0;
    GenInt(t,:) = randint(1,GeneNum,[StartValue,EndValue]);
    % check if there are any duplicated genes in this chromosome
    [BB index]=sort(GenInt(t,:));
    for tt=2:GeneNum
        if ( BB(tt-1) == BB(tt) )
            Flag(1,t) =Flag(1,t)+ 1;
            flagT=1;
```
% if there is not duplicated genes in this chromosome, get the next chromosome
if (flagT ~=1)
    t = t + 1;
end
end

% This function is to return two arrays
% SortDistNeighbors---- the distance among the samples
% NeighborIndexKNN ----- return the KNum nearest neighbors of the samples
% SamplesVectors holds the samples genes' expression values
% GenIntIndex is the index of the chromosomes array
% GenInt is the chromosomes array
% TraingSamples tells the training samples
% GenInt(GenIntIndex) is the d-dimension
% calculate the distance among the training samples, then find the nearest neighbor
% use the genes presented in GenIntIndex as genes to find out
% KNum neighbors for each sample
function [SortDistNeighbors,NeighborIndexKNN] = ...
    KNN(SamplesVectors,GenIntIndex,GenInt,KNum,TraingSample)

TraingLength = length(TraingSample);

for t=1:TraingLength
    for tt=1:TraingLength % gene position
        Dist(t,tt) = Distance(SamplesVectors,TraingSample(t),…
                              TraingSample(tt),GenIntIndex,GenInt);
    end
end

for t=1:TraingLength
    [SortDistNeighbors(t,:),NeighborIndex(t,:)] = sort(Dist(t,:));
    for tt=1:TraingLength
        NeighborIndex_1(t,tt) = TraingSample(NeighborIndex(t,tt));
    end
end

for t=1:TraingLength
NeighborIndexKNN(t,1:KNum) = NeighborIndex_1(t,2:KNum+1);
End

/**/******************************************************

MutationIntFunc.m

************************************************************************/

% This function is to find the gene position for mutation, GeneInt and Gene position
% replaced with the one (gene) from the gene pool, and the gene id is not in the Chro.
% in this function, maybe some rows in GeneOutInt matrix are zeros,
% because: in line 18 to 20, just assign some rows to GeneOutIndex,
% then the system will assign the rest rows to zeros ######
% changed!!!!, GenOutInt = GenInt ######

function [GenOutInt,MutationPosition] = MutationIntFunc(FitnessValue, …
    MutationRate,GeneNum,GenInt,StartValue,EndValue)

    [MutationPosition,Mupo,MutationValue] = MutationPoiFunc(FitnessValue, …
        MutationRate,GeneNum);

    GenOutInt = GenInt;
    size_mut = size(MutationPosition);
    size_gene = size(GenInt);

    for t=1: size_mut(1)
        k=0;
        while (k<size_gene(2))
            temp=randint(1,1,[StartValue,EndValue]) ;
            for tt=1:size_gene(2)
                if (GenInt(MutationPosition(t,1),tt) == temp)
                    k=0;
                    break;
                else k=k+1;
            end
        end
        GenOutInt(MutationPosition(t,1),MutationPosition(t,2)) = temp;
    end

    /************************************************************

MutationPoiFunc.m

************************************************************************/

function [MutationPosition,Mupo,MutationValue] = …
    MutationPoiFunc(FitnessValue, MutationRate,GeneNum)
CumuPro = CumuProbaFunc(FitnessValue);
Size_chrom = size(FitnessValue);
NumMutate = fix( Size_chrom(2) * GeneNum * MutationRate);
k = 0;
while (k==0)
    temp = rand(1,NumMutate);
k = 0;
    for t=1:NumMutate
        if(temp(t) <= MutationRate )
            k = k+1;            Mupo(k) = t;            MutationValue(k)=temp(t);
        end
    end
end

for t=1:k
    MutationPosition(t,1)= fix(Mupo(t) /GeneNum ) + 1;
    MutationPosition(t,2)= mod(Mupo(t) ,GeneNum );
    if(MutationPosition(t,2)==0)
        MutationPosition(t,2)= GeneNum;
        MutationPosition(t,1)= MutationPosition(t,1)-1;
    end
end

/***************************************************************************/
Preprocessing_aml_all.m
/***************************************************************************/

% This is the data processing function created for preprocessing the data set
% of aml_all, download from
% http://www.broad.mit.edu/cancer/software/genepattern/datasets/
% There are 38 samples in training dataset, 35 samples in testing dataset.
%
% 1. The combined training and testing dataset (which includes vector expression and
% quality lables as a pair for each gene and sample) are stored in traintest_tect.txt file,
% the genes names are stored in SecondColulmn.txt file.
% 2. the quality lables settings are stored in parameter.txt file which define P,M,A values
% and K value, which is threshold value. (parameter.txt: A:0, M:0.5, P:1, K:0.5 )
% 3. input files of this function
%    parameter.txt,trainrest_vec.txt, SecondColumn.txt
% 4. output files of the function
%    TrainData_05.txt: records genes quantative and quality values for all training samples.
%    TestData_05.txt : records genes quantative and quality values for all testing samples.
%    Data_05      : records genes quantative and quality values for training and
clear

Num_train = 38;  Num_test = 35;  Num_Samples = Num_train + Num_test;

[Names,Values] = textread('parameter.txt', '%s %f', 4);  % read in quality values
PValue = Values(1);  MValue = Values(2);  AValue = Values(3);  KValue = Values(4);
NumColumns = 2*Num_Samples;

fp=fopen('traintest_vect.txt', 'r');
fp_title = fopen('SecondColumn.txt', 'r');
fp_train = fopen('TrainData_05.txt', 'w');
fp_test = fopen('TestData_05.txt', 'w');
fp_data = fopen('Data_05.txt', 'w');
fp_name = fopen('Name_05.txt', 'w');

line = fgets(fp);  % read one line from data set
line_name = fgets(fp_title);  % read in one for the name of the gene

line_num=1;  y_num=0;
while (line > 0)
    s=line;
    num_As=0;  num_Ms=0;  num_Ps=0;
    for t=1:NumColumns
        [token rem] = strtok(s);
        s=rem;  num_float(t) = length(token);
        if(num_float(t) > 1)  values(t) = str2num(token);
        else
            if(strcmp(token,'P')==1)  values(t)=PValue;  num_Ps = num_Ps +1;
            elseif(strcmp(token,'M')==1)  values(t)=MValue;  num_Ms = num_Ms +1;
            elseif(strcmp(token,'A')==1)  values(t)=AValue;  num_As = num_As +1;
            else  values(t)=0;
        end
    end
    temp_sum =0;
    for t=2:2:NumColumns
        temp_sum = temp_sum + values(t);
    end
    line = fgets(fp);
end

for t=1:NumColumns
    line = fgets(fp);
end
average = temp_sum / Num_Samples;

if (average >= KValue)
    y_num = y_num+1;
    for t=1:2:NumColumns
        fprintf(fp_data,'%12.6f',values(t));
        fprintf(fp_data,'%5.2f',values(t+1));
    end
    fprintf(fp_data,'\n');

    for t=1:2:Num_train*2
        fprintf(fp_train,'%12.6f',values(t));
        fprintf(fp_train,'%5.2f',values(t+1));
    end
    fprintf(fp_train,'\n');

    for t=Num_train*2+1:2:NumColumns
        fprintf(fp_test,'%12.6f',values(t));
        fprintf(fp_test,'%5.2f',values(t+1));
    end
    fprintf(fp_test,'\n');

    fprintf(fp_name,'%6d %6d %6d %6d %6d
', line_num,y_num,num_As,num_Ms,num_Ps);
    length_title = length(line_name);
    for t=1:length_title-1
        fprintf(fp_name,'%c',line_name(t));
    end
    fprintf(fp_name,'\n');
end

clear token, rem, s ;

line = fgets(fp);
line_name = fgets(fp_title);
line_num = line_num +1;
end
fclose(fp_name); fclose(fp_test); fclose(fp_train); fclose(fp_data); fclose(fp);
close(fp_title);

******************************************************************************
SaveResultsFunc.m
******************************************************************************

function [BestNum,FitnessValueFunc,SortedIndex,GenResultsInt]= ...
    SaveResultsFunc(GenInt,ChromoNum,KNum,SamplesVectors,...
for t=1:ChromoNum
    [FitnessValueFunc(t),SampleScores(t),TS(t,:)] = FitnessIntFunc(t,GenInt,KNum, …
    SamplesVectors,SampleClass, TraingSample,SamplesQuiltVector,partialFlag) ;
end
[SortedFitness, SortedIndex ] = sort(1-FitnessValueFunc);

size_Gen= size(GenInt);  Num_samples = length(TraingSample) ;
adjust=0;  t=1;
while (FitnessValueFunc(SortedIndex(t)) >= (Measure-adjust))
    for tt=1: size_Gen(2)
        fprintf(FId,'%6d',GenInt( SortedIndex(t),tt) );
    end
    fprintf(FId,'--- %f
',FitnessValueFunc(SortedIndex(t)) );
    GenResultsInt(t,:)=GenInt( SortedIndex(t),:);     t=t+1;
end
BestNum=t-1;

\**********************************************************************
TestFitnessFunc.m
***********************************************************************/
function [FitnessValue,SampleScores,T] = TestFitnessFunc(GenIntIndex,GenInt, …
    KNum,SamplesVectors,SampleClass,TraingSample)
    [SortDistNeighbors,NeighborIndexKNN] = …
    KNN(SamplesVectors,GenIntIndex,GenInt,KNum,TraingSample);
    TraingLength =length(TraingSample);
    for t=1:TraingLength
        T(t) = AgreeFunc(NeighborIndexKNN,t,KNum,SampleClass,TraingSample) ;
        if(T(t)==KNum)  Score(t)=1;
        else            Score(t)=0;
        end
    end
    FitnessValue = sum(Score)/TraingLength;
    SampleScores = sum(Score);

\**********************************************************************
TopGeneSelectFunc.m
***********************************************************************/
function GeneFrencyResult =  TopGeneSelectFunc(GenTotalResultsInt,…
RangNum, fitnessScores, SamplesQuiltVector, TopGenNum)

[ChromoNum, chromoLength] = size(GenTotalResultsInt);
[SampleNum, geneLength] = size(SamplesQuiltVector);

t = 1:1:RangNum; GeneFrequencyN = hist(GenTotalResultsInt, t');

for tt = 1:RangNum
    GeneFrequencyTemp(tt, 1) = sum(GeneFrequencyN(tt, :));
end

k = 0;
for kk = 1:RangNum
    if (GeneFrequencyTemp(kk) ~= 0)
        k = k + 1;
        GeneFrequency(k, 1) = GeneFrequencyTemp(kk); % the gene's frequency
        GeneFrequency(k, 2) = kk; % the gene id
    end
end

for t = 1:k
    temp = 0;
    for tt = 1:SampleNum
        temp = temp + SamplesQuiltVector(tt, GeneFrequency(t, 2));
    end
    avgQuilt(1, t) = temp / SampleNum;
    TempGeneFrequency(t, 1) = avgQuilt(1, t) * GeneFrequency(t, 1);
    TempGeneFrequency(t, 2) = GeneFrequency(t, 2); % gene id
end

[freqsort, freqindex] = sort(TempGeneFrequency(:, 1));
sortlistlen = length(TempGeneFrequency);

for t = 0:sortlistlen - 1
    GeneFrequencyResultTemp(t + 1, 1) = TempGeneFrequency(freqindex(sortlistlen - t), 1);
    GeneFrequencyResultTemp(t + 1, 2) = TempGeneFrequency(freqindex(sortlistlen - t), 2);
end

GeneSelectedNum = k;
GeneFrequencyResult(1:TopGenNum, 1:2) = GeneFrequencyResultTemp(1:TopGenNum, 1:2);