Executive summary

Classifying patients into the correct subtypes of cancer is one of the most important challenges faced by cancer researchers. Correct classification means that patients can receive treatments more suited to their specific needs so that they don't have to go through unnecessary treatments and side effects.

Our goal in this project was to write a program that would separate patients into one of two categories of leukemia by analyzing their gene expression levels. The intent of our project was the simple classification of two leukemia subtypes, however, our program has been written so that it can be easily extended to the classification of more subtypes.

Leukemia is a type of cancer that affects developing blood cells in a patient's bone marrow. There are as many as 150 subtypes postulated to exist. In this project our goal was to distinguish between two important leukemia subtypes: acute myeloid leukemia (AML) and acute lymphoblastic leukemia (ALL). Though AML generally is found in adults and ALL is a childhood cancer, it is not always possible to classify which subtype a patient has simply on the basis of age. It is not even possible to perfectly classify the patients based on lab assays. Researchers want to be able to distinguish between AML and ALL at the genetic level.

The Whitehead group at MIT has recently performed numerical experiments to classify leukemia, and we obtained the data set they analyzed and the results of their model, which we compared with our results using a different model. Their data set contained bone marrow samples from 38 leukemia patients with AML or ALL. The data set was analyzed using microarray gene expression analysis, which determines the levels of gene expression for a given patient sample for thousands of genes. They originally had 6817 genes to use in identification, but they reduced this number to 50 genes that seemed related to AML and ALL. We used this same training data set of 38 patients and 50 genes to train our own program; we also used the same test data set containing 33 patients.

Our neural network is designed to classify the patients; it includes parallel processing to divide the computational work between many linked processors. This was an experiment in classification techniques; though many techniques have been explored, including well-known statistical methods and newer methods such as genetic algorithms, as far as we know there have been no other attempts to apply neural networks to this type of classification problem.

The neural network implemented in our program takes in the numerical gene expression values and uses them to determine leukemia subtypes. There are three layers in our neural network: input, hidden, and output. The data set is stored in the input and continues on to the hidden layer. Between the input and hidden layers there are weights that change the numerical values and then the new values are plugged into a function. The same process is repeated with different weights between the hidden and output layers, and the final output gives the identification number of the sample and identifies it as AML or ALL.

Before we can run actual data through our neural network, we use backpropagation to train the neural network and optimize the weights. In backpropagation, we run a training data set through the neural network and compare the output of the neural network to the actual training output to measure the amount of error. The program then runs through backwards and adjusts the weights to reduce the error. The training data is run through repeatedly, going forward to test the neural network and then backwards to make adjustments. Finally, the margin of error is minimized and the neural network is fully trained. At that point, we can use the neural network to try to classify a test data set. This is a data set for which we know the results, but the neural network does not. Therefore, we can compare the results of the neural network to the real results to test whether the program is trained.

Once we had written the neural network code, we added message-passing interface commands to parallelize the training section of our program. When our program runs in parallel, different processors compute weight updates and contributions from different patients and the end results are communicated between the processors. In this way, we can run through the data more quickly, especially for large training data sets.

We are continuing to run trials through our neural network and analyze the numerical results. When we have completed our tests the code we developed will continue to be used in part and in its entirety by the HPCC researchers and graduate students to classify leukemia patient data.

Introduction

The title of our project is “Parallel Processing of Genomic Leukemia Data Using Neural Networks.” Our program analyzes genetic information of leukemia patients and runs the numeric levels of gene expression through a neural network to classify the patients into different subtypes of leukemia. The two leukemia subtypes that we are distinguishing between are AML (acute myeloid leukemia) and ALL (acute lymphoblastic leukemia). We chose to write our program to run in parallel in anticipation of large data sets. Though parallelization is unnecessary for our current data set (38 by 50), the eventual size of data sets used by researchers will be very large, comprising data from tens of thousands of patients and tens of thousands of genes.

Our project is related to a larger project that is currently being worked on by the UNM School of Medicine and the Albuquerque High Performance Computing Center. They are trying to find specific connections between patients’ gene expression levels and their subtypes of cancer.

When we started this project we were intrigued by the possibility of programming and parallelizing a neural network; we also wanted to write a program relating to current science. We chose this particular problem because it concerns the emerging area of medical science surrounding the Human Genome Project. Since the human genome was only recently mapped and published, there is much yet to be discovered in this research field and we see an opportunity to contribute to this work. We have all known people who have had cancer or been affected by it, and so we want to do anything we can to help cancer research.

Background

Leukemia

Leukemia is a cancer of developing blood cells in the bone marrow. It causes abnormal, immature cells to multiply, which interferes with the production of healthy cells in the normal bone marrow. This results in a lowered immune system, putting the patient at risk for minor infections [4].

Leukemia affects about 35,500 new individuals each year in U.S., and 34,000 people are estimated to die of the disease each year, 6300 from AML and 1410 from ALL. There are approximately 150 distinct genetic subtypes believed to exist, each associated with specific abnormal chromosome recombination and defined at the genetic level. Two important subtypes of leukemia are AML (acute myeloid leukemia) and ALL (acute lymphoblastic leukemia). AML predominantly occurs in adults and with aggressive therapy 70-80% of patients achieve remission and 25-30% are cured. ALL predominantly occurs in children and is the most common form of childhood cancer; with aggressive therapy, 80-90% of patients achieve remission and 70-80% are cured. Both AML and ALL can be fatal within months if they are not treated [4].

Though both ALL and AML are severe types of cancer, researchers usually focus on ALL because it almost exclusively strikes children. A short-term goal of our program is to distinguish between AML and ALL so that researchers will be able to classify the ALL patients into more specific subtypes. However, the ultimate goal is a more detailed classification overall.

Gene Expression Microarray Analysis

The classification of leukemia patients into AML or ALL is currently being done by trained physicians, which produces an unacceptably high margin of error. This is not due to limitations in their skills or training but to the similarity between different subtypes of leukemia. Distinction between different subtypes is understandably difficult, considering that the differences are often so subtle that they can only be detected at the sub-cellular or genetic level. One test that can be performed to detect genetic subtleties such as these is gene expression microarray analysis.

In order to obtain the specific genetic information required to distinguish patients from one another at the genetic level, scientists use a methodology known as gene expression profiling. In this technique they measure the relative expression of a gene in various patients using microarray analysis. Scientists obtain blood or tissue samples from leukemia patients. The messenger RNA (mRNA) within each sample, which reflects the extent to which a gene is expressed or activated within a cell, is then extracted and tagged with a fluorescent dye. A different dye is used to tag a reference sample. The mRNA patient and reference samples are analyzed using a probe array or gene chip. The probe array consists of a slide of membrane containing microscopic wells, each containing a section of DNA corresponding to a particular gene.

The gene chip is bathed in a sample in order to analyze it. The mRNA within the sample that correspond to a given section of DNA on the chip then hybridize or attach to them. The chip then is scanned with a laser and researchers detect the fluorescence intensities of the hybridized mRNA. The fluorescence ratios of the leukemia patients relative to that of the control sample

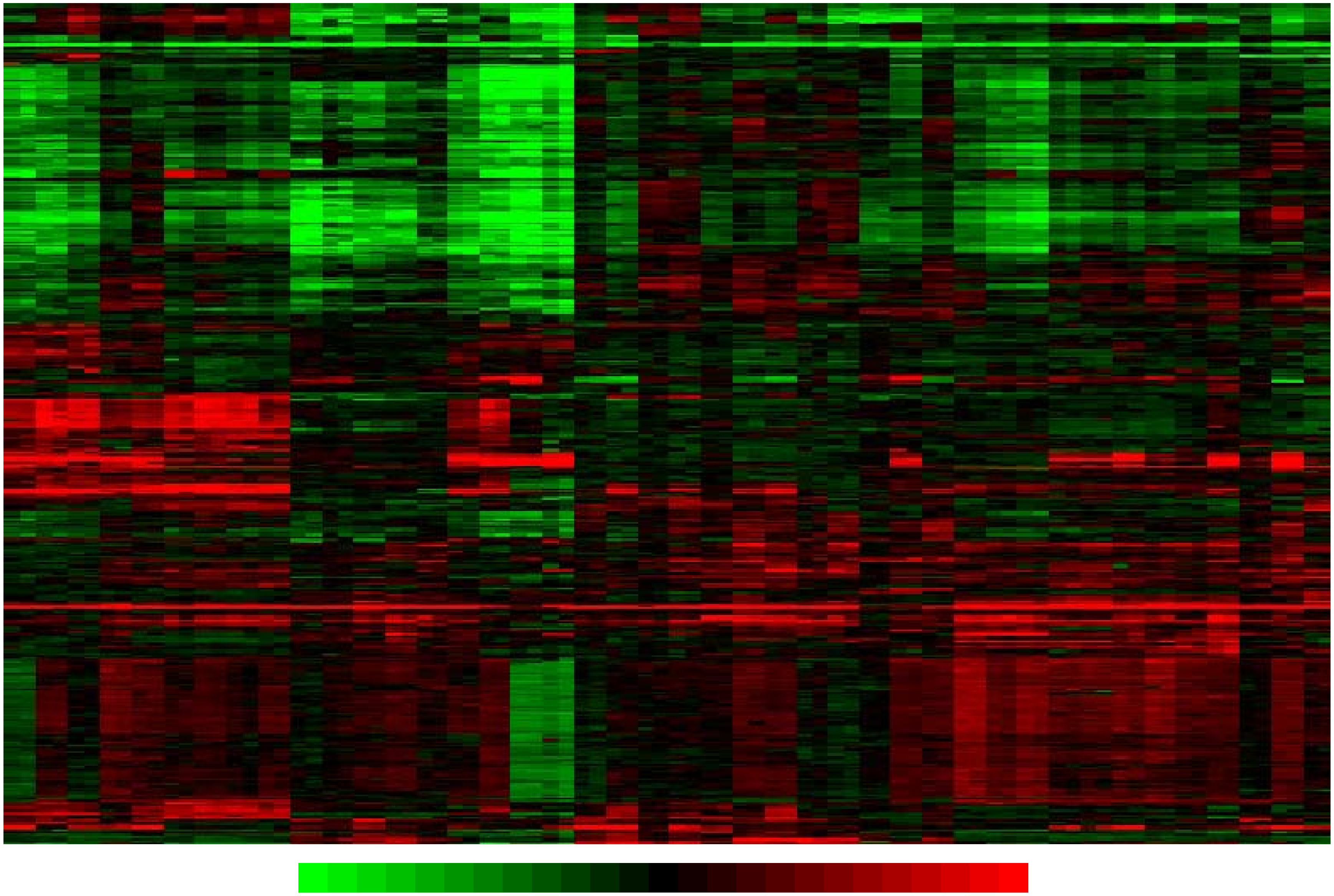

Figure 1. This is a visual representation of the gene expression values for each patient.

are computed and the final image displays the levels of gene expression for each patient for each gene represented on the chip.

This image is displayed using different fluorescent colors: red, green, brown or yellow, and black. In Figure 1, the red fluorescence corresponds to the leukemia patients and green to the reference. Brown fluorescence represents an overlap of both leukemia and reference samples. Black represents genes that weren’t expressed in either sample. The different shades of these colors represent the different fluorescence intensities. The gene intensities are available in numeric form, which we can use as input to our program.

Whitehead Group

The Whitehead research group at MIT recently conducted numerical experiments to classify leukemia patients into one of two subtypes (AML or ALL) by analyzing their gene expression levels, just as we are doing in our project. In fact, we are using the same training and test data sets that they used, though our classification methods are different. They posted their data set and results on the Internet for use by the cancer research community, with no restrictions, and we have used their results to train and test our neural network, as well as to compare the result of our methodology with theirs.

The Whitehead group used a training data set consisting of 38 bone marrow samples from leukemia patients, 27 of whom had ALL and 11 of whom had AML [5]. On the Affymetrix gene chip they used there were probes for 6817 human genes [5, 7], and patients’ expression levels were obtained for each gene. The Whitehead researchers then selected fifty of those genes from a larger subset of several hundred genes that they had identified as “optimal” in distinguishing between AML and ALL in their test data set. They reduced the number of genes to avoid overfitting the data and thus avoid measuring excess noise and not the actual results.

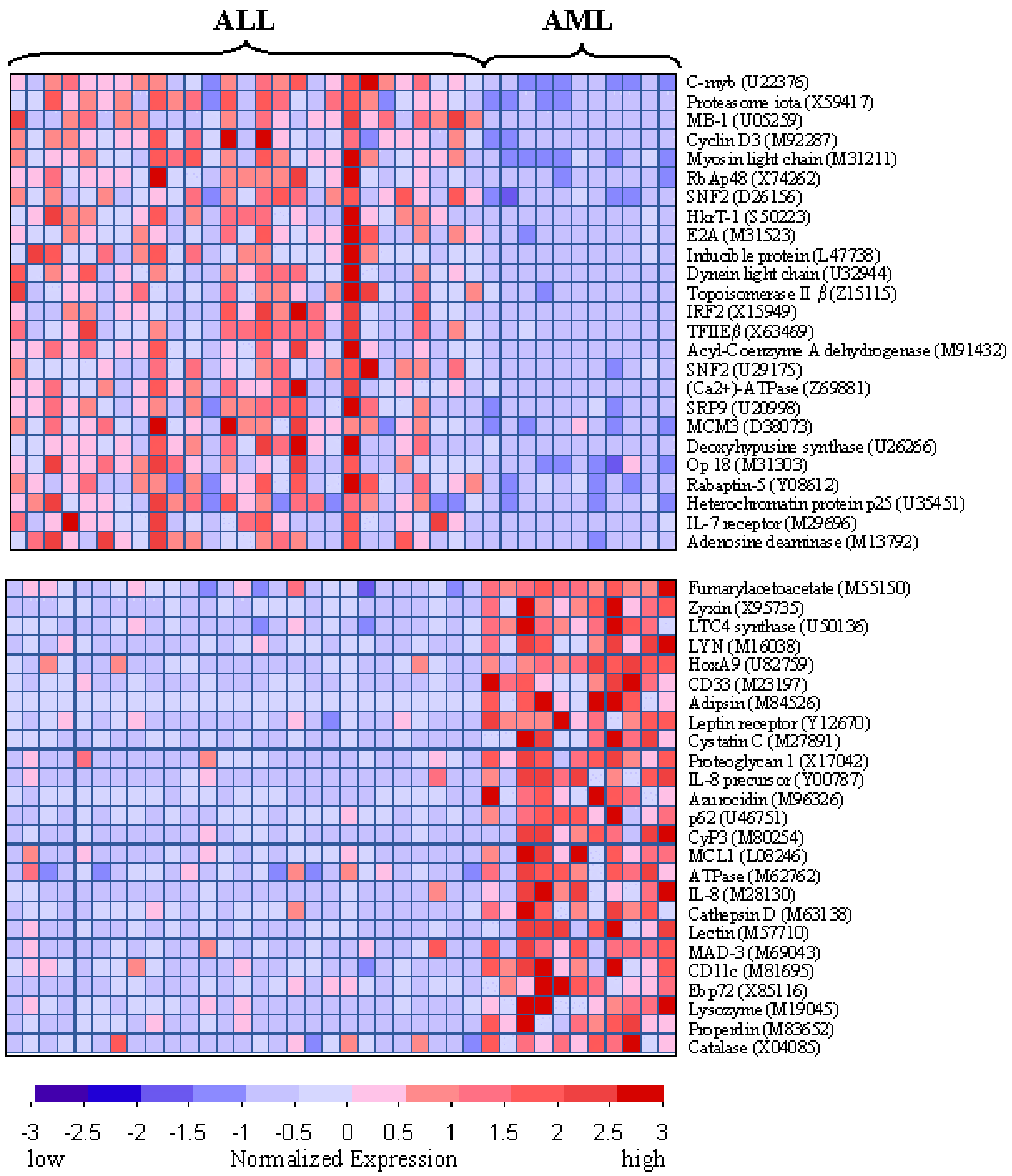

Figure 2. This is the visual representation of the training data set used by the Whitehead group and by us. Each column is a different patient from the set of 38, and each row is labeled with one of the 50 genes. The intensities of the fluorescence colors, shown on the bar below, represent the levels of gene expression.

Feature Selection vs. Feature Extraction

The fifty genes were chosen using feature selection, in which the features that best characterize a given pattern are selected from a larger set (in our case, a subset of gene expression levels for a given patient). In selecting the fifty genes, the Whitehead group looked for genes that were "strongly correlated with the class distinction to be predicted" [5]. They wanted genes that displayed different expression levels for the two classes, but that gave consistent results within each class. They determined this with a method they called "neighborhood analysis," in which they compared the genes to an "idealized expression pattern" that was uniformly high in one class and low in the other, and found the genes that displayed a similar pattern. Basically, the selected genes displayed significant variation in expression levels due to the class distinction [5, 7].

An alternate method that is often used is feature extraction, which "is a more general method in which the original set of features is transformed to provide a new set of features" [6]. In feature extraction, genes with the same patterns of fluorescence are grouped together and become a single feature, which reduces the total number of effective features [1, 6]. This is more complicated than feature selection, but it may be more accurate.

Neural Networks

A computational neural network (NN) is made up of "neurons" that are interconnected by weights [3]. Neural networks can be used in many types of pattern recognition problems; here, we apply them to the problem of recognizing patterns of gene expression that characterize the two different types of leukemia we are trying to distinguish; AML or ALL. To perform the necessary calculations in our program we used standard NN equations that we obtained from the book Pattern Classification by Richard O. Duda, Peter E. Hart, and David G. Stork [2] [see Appendix A].

Between the input and output layers in a NN, there can be several layers that are called "hidden" because they exist between the only layers with which the user interacts (to input data and receive output) [2]. There are three layers in our NN: an input layer, a hidden layer, and an output layer; the ratio of nodes in each is $50{:}n_H{:}1$, where n_H is the number of hidden nodes, a user-controlled variable. There are 50 input nodes corresponding to the 50 selected genes. Since there are only two output categories, AML and ALL, only one output node is necessary to distinguish between them.

Connecting each set of layers are weights that influence the data being processed through the NN. A data set is entered as gene expression intensities and is stored in the input layer. Then it is multiplied by the weights between the input and hidden layers and the product is plugged into a net activation function that passes its answer to the hidden layer. This process repeats between the hidden and output layers with different weights, and this determines the output for each patient: AML or ALL.

Before a NN can process real data it must be trained using test data. The training method we have used is backpropagation, which works well on a feed-forward network such as ours [8]. Once the test set is run through the NN (feed-forward), the output is compared to the target output, and then the backpropagation begins. In backpropagation, the program goes through the NN backwards to adjust the weights so that the error between the feed-forward output and the target output is reduced. This process is repeated until the feed-forward output matches the target output to within a user-specified error tolerance.

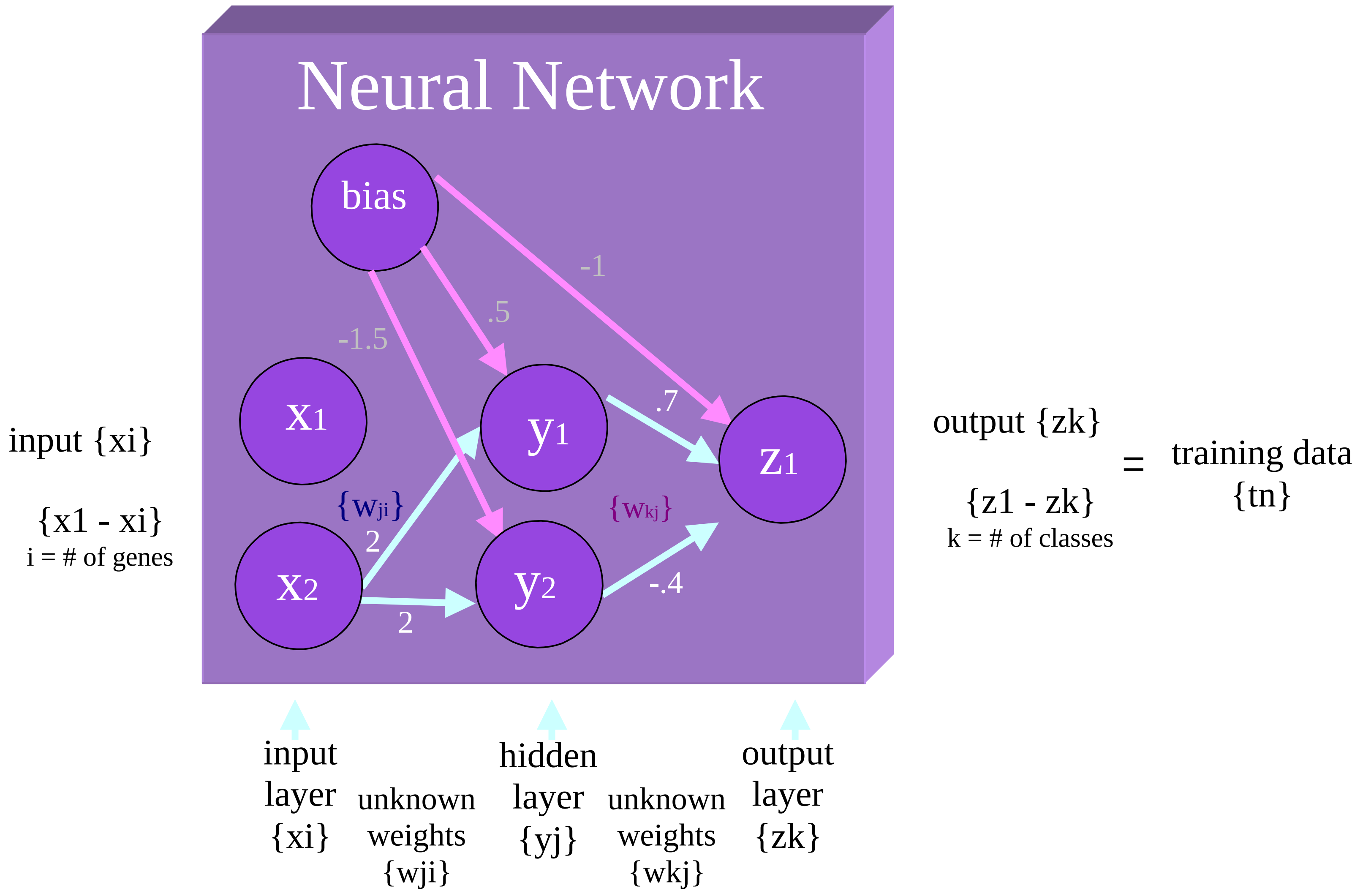

Figure 3. This is a diagram of our neural network. The circles represent the nodes at each of the three layers, and the arrows show the path of data through the network.

There are different algorithms for backpropagation, including stochastic and batch. Our program uses both stochastic and batch because certain parallel processing algorithms only work with certain types of backpropagation, and we wanted our program to be able to run both in parallel and on a single processor. We also wanted to compare the two approaches.

When our program uses batch training, the patients are run through the program one by one in numerical order and the weights are changed after all the patients have gone through. In stochastic training, patients are randomly selected from the training set and the weights are readjusted after each individual patient.

After the backpropagation has run through repeatedly and adjusted all the weights until the output has reached the target within the specified error tolerance, the neural network has been fully trained and the weights are fixed at their final values. The test data set is then run through the neural network to determine whether a set of patients is correctly classified into either AML or ALL. Now, in theory, this neural network with these weights could be used by researchers to classify an entirely new set of patients.

Parallel Processing / MPI

Parallel processing is a programming technique that allows multiple processors to work together on a given problem simultaneously. This lets the program process data more quickly and provides more memory in which to store large data sets, since the data can be divided among processors. For this reason it is often used for problems that are complex and/or involve large data sets. Since this area of research is expected to involve increasingly large data sets, it made sense for us to parallelize our program. Also, the nature of a neural network makes it well suited to parallization. Our personal motives in parallelizing our neural network were that we wanted to write in parallel as a personal challenge and to increase our knowledge of more advance programming techniques.

The set of functions that we are using to implement parallelism in our code is called the Message-Passing Interface (MPI).

There are two basic approaches to the parallelization of a neural network: *fine-grained* parallelism and *training set* parallelism. In fine-grained parallelism the actual neural network is partitioned and the different parts of the code corresponding to the different parts of the neural network are run on separate processors. This style of parallelism works best for a program in which the neural network is very complex, as measured by the number of nodes and weights. Ideally, fine-grained parallelism is used when the complexity of the neural network exceeds the number of training sets. However, fine-grained parallelism requires a lot of communication between processors because it passes specific pieces of information to specific processors repeatedly in the program.

We chose to use training set parallelism because it is more straightforward than fine-grained parallelism and this is the first time any of us have parallelized a program. In training set parallelism each processor runs a complete version of the code on different data sets or different portions of the same data set. The same program is run on all processors simultaneously and they all update the weights independently. This style is more useful when the neural network is less complex than the number of training sets, as in our program. One advantage of training set parallelism is that it requires less communication between the processors and so can work faster on some problems than fine-grained parallelism. In training set parallelism, MPI is used to distribute the program to all processors and they work on it separately, only communicating after a given training iteration, when they pass on their weight updates to one processor and it adds them together.

Fine-grained parallelism works with any kind of backpropagation, whereas training set parallelism only works with batch backpropagation. Though we were limited to using batch training for the parallel version of the program, we also included stochastic training in case the program is run on a single processor.

Description

Code Description

Our program is a complete modeling program in C that enables us to try out different computational models in the form of a neural network with variable parameters to see which one works best for our problem. Our code uses the gene expression information obtained from the microarray data sets made available by the MIT Whitehead group on the World Wide Web (gene expression values for several sets of patients across a set of 7129 distinct DNA probes) to classify a training set of 38 leukemia patients into AML or ALL. This is done by feeding the gene intensities, appropriately scaled, into the neural network. A standard neural network training cycle is performed, yielding a converged set of weights. These weights are then fixed and applied to the classification of the test data set. For those interested in a step-by-step description of what our code does, along with the rationale underlying our coding approach, see Appendix B. For a copy of the program itself, see Appendix C. For the other components of the program (the Makefile used for compilation, timers.c, nntimers.h, nnclass.c, nnclass.pbs, nnclassparam.dat), see Appendix D.

Practical Application

Our completed program can be applied to any data set of cancer patients to classify them into two known subtypes. Although the input section of our program our program was written to use fifty specific genes to classify patients into AML or ALL, the neural network core of the program is entirely general, and the code can be easily modified to use a different gene input set and to classify into an arbitrary number of different subtypes.

Our program also uses dynamic array allocation, rather than static array allocation, which means that the user can input values for the number of genes, the number of patients, the number of hidden nodes, etc., at runtime instead of having them hardwired into the program. This is considered to be a good programming technique because it allows the user to define the computational environment and it prevents the necessity of compiling the program for each change in numerical parameter values.

There is only one major limitation to our program: since our program is designed to do class prediction, in which patients are assigned to known classes, it cannot identify new classes (class discovery) [5, 7]. However, class discovery is beyond the scope of this project, and we wrote our program with the knowledge that it is limited to previously discovered subtypes.

The success of our program ensures that it will be used and further developed by researchers at the Albuquerque High Performance Computing Center (AHPCC) and the University of New Mexico School of Medicine (UNM SOM). Our mentor, Dr. Atlas, will give our code to her graduate students and have them work with it. They will try to use different genes to do the classification and they will change the number of genes used (we used fifty). Our program will for the basis for further research into the application of neural networks to the molecular classification of cancers.

Also, the UNM SOM will use our program to analyze their own data in the future; currently, the labs are still being set up and experiments have not yet begun. However, once the labs are prepared, the SOM researchers will begin a series of gene expression experiments. Cheryl Willman, M.D., the Director and CEO of the Cancer Research and Treatment Center at the UNM SOM, has obtained a data set of 120 infant leukemia samples, and our program will be used to classify them in terms of favorable or non-favorable outcomes. These two outcome possibilities are analogous to this project's classification of samples into AML or ALL. Later, our program may be used on even larger data sets.

Researchers want to know if there is a correlation between a patient's genes and subtype of cancer, and our program will help them find out. However, the importance of cancer classification is not only for research purposes, but also for accurate treatment. By knowing the exact subtype of cancer a patient has, doctors can choose the treatment regimens that will be most effective. This improves treatment accuracy and lets patients avoid unnecessarily severe treatments and the resulting side effects.

Materials

Our program was written in the language C, using Message-Passing Interface (MPI) functions to implement the parallelization. As mentioned before, we used a data set from the Whitehead group at MIT to train our neural network.

We developed our program on the Truchas workshop cluster at the AHPCC, which contains twelve dual-processor machines that are generally available to researchers. Each processor is a 650 MHz Pentium III, with 256 megabytes of RAM.

Once we parallelized our program we began to run it on the AHPCC "Linux supercluster," BlackBear. A Linux supercluster is a supercomputer built from individual PCs, each of which runs its own copy of the Linux operating system. The PCs in BlackBear are linked by an ultra-fast commercial network interconnect. BlackBear is made up of 16 dual-processor symmetric multi-processor (SMP) nodes; each processor is a 550 MHz Pentium III.

Since we used the machines at the AHPCC, we had to use the Linux operating system to access our program and manipulate our files. We used the text editor *vi* to write our program, mostly for its usefulness in debugging. When the compiler found an error, we could command vi to jump to that specific line. The two compilers we mainly used were *gcc* (a GNU compiler for C) and *pgcc* (a compiler for C from the Portland Group). To compile the parallel processing, we used *mpicc* (a PGI compiler with the MPI library).

Results

The result of our project is a general, extendable neural networking program. Our code was written to be flexible so that it is not specifically tailored to the data set we have. It allows the user to determine the parameters of the neural network so that it can be used on all different sizes of data sets. Also, our code can be extended to use the expression levels from more than fifty genes in classification. The result of this flexibility is a complex code with the capability to classify virtually any data set from this field of research.

Another effort we have made to expand the capabilities of our program is the option of parallelization, which we have included by instrumentation of parallel subroutine calls.

Due to the nature of modeling programs such as our own, it takes many trials to obtain numerical results. Though we have a working code, we have yet to produce significant numerical results, and so we will spend the next two weeks running data through our completed program in order to obtain them. We have two data sets we are feeding program, a training data set and a test data set, and our code will provides the numerical results for each.

Knowledge Acquired

In the course of this project we learned more about the science involved, especially leukemia and microarray gene expression analysis. Though we all had a basic idea of what leukemia was prior to this project, we have gained more in-depth knowledge concerning the different subtypes AML and ALL. Everything we now know about microarray gene expression analysis we learned while researching and working on this project.

We also learned some entirely new programming methods and commands. This was our first experience in writing a neural network, and the fact that we were doing pattern recognition made it even more of a challenge. This was also the first time any of us had attempted to write a code using parallel processing, and so we had to learn all about how they worked and how to incorporate them into our code. We also had to learn the commands for the text editor vi and some Unix to effectively utilize the operating system on the machines at the AHPCC.

We sharpened our previously acquired skills in programming in C, writing reports, and giving presentations as well. However, we got more out of this project than just new knowledge; we also developed stronger friendships and practiced working together as a team.

Conclusion

Neural networks have been used in pattern classification for years, but they are only recently being applied to this type of genomic problem in the research community. Now that a working NN program exists to classify leukemia data into two subtypes, the effectiveness of this computational approach can be quantified and compared to other approaches.

A collaboration between the AHPCC and the UNM SOM is planning to utilize techniques such as the ones used in our program for their large-scale genomic leukemia research. In fact, they will use our program to aid in their research. Our mentor will give our code to some of her graduate students, one of whom is actually conducting research in this field for his PhD thesis, and have them use our neural network as a basis for their own similar research. Our code will be used to conduct extensive parameter studies and test the application of neural networking techniques to genetic classification problems. Our neural network is slated to be incorporated into later codes, and until then it will be used on actual patient data sets.

Our program will, in its own small way, contribute to current research on classification techniques, and test the reliability of neural networks in identifying the different subtypes of leukemia.

Recommendations

Our project required us to narrow our field of interest, so that we would be able to accomplish our goals in the allotted amount of time. Initially, we had discussed using genetic programming methods, but as our project moved forward, it became apparent that this technique would not be useful in solving our problem.

We had anticipated that we would be able to display our results with VxInsight, a 3D visualization tool developed by Sandia National Laboratories, which has been recently used in certain types of microarray data analysis. Unfortunately, VxInsight is not designed for the type of data used in our project, and our mentor did not discover this until recently.

We had also hoped to use a larger data set than the MIT data; we planned on using samples gathered by the UNM SOM from patients who had participated in several recent clinical trials. This was not possible because the microarray facility at UNM that will be used to analyze the samples has been replanned during the course of our project and is still in the process of being set up, which prevented us from incorporating this newer data into our project. However, our code will be passed on to the graduate students in the computational group at UNM, some who plan to use parts of it in their own investigations of computational approaches to leukemia data classification, and others who plan to apply our code directly to the analysis of the UNM SOM data.

Acknowledgments

We would like to thank everyone who helped us during the course of this project; we could never have finished our project without all their input.

Thanks go to our mentor from last year's supercomputing challenge, Dr. Larry Sklar, who gave us the idea for this year's project. He suggested that we look into the cancer research program at UNM, and he contacted Cheryl Willman via email. Thanks also to Cheryl L. Willlman, M.D., the Director and CEO of the Cancer Research and Treatment Center at UNM, who provided us with information, invited us to sit in at science meetings, and arranged for us to meet our mentor, Susan Atlas.

Special gratitude goes to our mentor Susan R. Atlas PHD of the AHPCC. She selflessly mentored us throughout the project, meeting us two to five days a week. She donated countless hours of her time to us, both as a mentor and as a friend. Dr. Atlas provided us with an opportunity to work on the same project that she has assigned to her graduate students and she had faith in us that we could do it. She also provided the use of her personal timers for the parallelization, PBS script, Makefile, and resized our input data. She also provided research materials, including science papers and programming tutorials, and explained all aspects of the background information that we didn't understand – repeatedly, if necessary. She was also an excellent source of programming knowledge; she introduced us with more advanced debugging techniques, parallelization, and methods for resizing input data. When we had a problem with the code, she was ready to guide us through it. She never just told us the solution; she made us work to figure it out. However, her part in our project was not only related to research and programming. She also took us out for coffee, chatted with us about her life, and answered our personal questions. We learned about kosher, the Jewish Saturday Sabbath, and her opinions on programming languages (though we still prefer C to Fortran!). Dr. Atlas who works hard for everything that she is involved in and yet still finds time to build and maintain meaningful relationships with her colleagues, graduate students, and even us. Her compassion and altruism touched our hearts. Thank you!

We would like to thank the AHPCC for allowing us to use their facilities, for giving us assistance, for letting us in early, letting us stay late, and for giving us candy. Thanks also to Dr. Andrew C. Pineda, Dr. Bob Ballance, Jim Prewitt, Dr. Ben Jones, Mark Fleharty, and other random programmers who made the mistake of wandering too close to us during programming sessions for their spot reviews of our code. Also, we would like to thank Patricia A. Kovatch for the use of her computer and for identifying people around the AHPCC.

Thank you to Gina Fisk, Betsy Frederick, and Mike Topliff of the NMHSSCC consult group for answering our questions about the report, and also thank you to all of you who are involved in the challenge; we appreciate the time you spend so that we can experience opportunities like this. Thank you for giving us advice and for reading our reports, even the lengthy acknowledgements section.

Lastly, we would like to thank the people who supported us in our efforts from the beginning: Mr. Neil D. McBeth and our parents. Mr. McBeth is our team sponsor, and he made sure that we made deadlines, attended all of the conferences, and had fun. He also provided transportation, food (peanuts!), and lots of stupid jokes. Thank you, Mr. McBeth, for letting us watch movies in your van, have a gum-chewing contest, and raid your “secret” candy stash. As for our parents… how can we thank them enough? They gave us food, shelter, and transportation, as parents are supposed to, but they took these even farther; they provided meals on the go, they shuttled us between supercomputing meetings, Tae Kwon Do, and music rehearsals at all times of day, and they gave us places to collapse and take naps when we were exhausted. Thanks for believing in us.

References

1. Christopher M. Bishop. Neural Networks for Pattern Recognition, Oxford University Press: New York, 1995.

2. Richard O. Duda, Peter E. Hart, David G. Stork. Pattern Classification, John Wiley & Sons, Inc.: New York, 2001.

3. Alex A. Freitas and Simon H. Lavington. Mining Very Large Databases With Parallel Processing, Kluwer Academic Publishers: New York, 1998.

4. Rose A. Gates and Regina M. Fink. Oncology Nursing Secrets, Hanley & Belfus: Philadelphia, 1997.

5. T. R. Golub, D. K. Slonim, P. Tamayo, C. Huard, M. Gaasenbeek, J. P. Mesirov, H. Coller, M. L. Loh, J. R. Downing, M. A. Caligiuri, C. D. Bloomfield, E. S. Lander. "Molecular Classification of Cancer: Class Discovery and Class Prediction by Gene Expression Monitoring," *Science* **286**, 531 (1999).

6. Michael L. Raymer, William F. Punch, Erik D. Goodman, Leslie A. Kuhn, Anil K. Jain. "Dimensionality Reduction Using Genetic Algorithms," *IEEE Trans. Evol. Comp.* **4**, 164 (2000).

7. Donna K. Slonim, Pablo Tamayo, Jill P. Mesirov, Todd R. Golub, Eric S. Lander. "Class Prediction and Discovery Using Gene Expression Data," preprint (1999).

8. Xiru Zhang, Michael McKenna, Jill P. Mesirov, David L. Waltz. "The backpropagation algorithm on grid and hypercube architectures," *Parallel Computing* **14**, 317 (1990).

Appendix A: Neural Network Equations

Feed-Forward

1. Net activation (input-hidden)

$$net_j = \sum_{i=0}^{d} x_i w_{ji}$$

2. Synapse (input-hidden)

$$y_j = f\left(net_j\right)$$

3. Net activation (hidden-output)

$$net_k = \sum_{j=0}^{n_H} y_j w_{kj}$$

4. Synapse (hidden-output)

$$z_k = f\left(net_k\right)$$

5. Activation function

$$f\left(net\right) = a \tanh\left(bnet\right) = a\left[\frac{1-e^{-bnet}}{1+e^{-bnet}}\right] = \frac{2a}{1+e^{-bnet}} - a$$

Backpropagation

6. Training Error (Least Squares Error)

$$J\left(w\right) \equiv \frac{1}{2}\sum_{k=1}^{c}\left(t_k - z_k\right)^2$$

7. Sensitivity of k

$$\delta k = \left(t_k - z_k\, f'\left(net_k\right) y_j\right)$$

8. Update of weights

$$\mathbf{w}\left(m+1\right) = \mathbf{w}\left(m\right) + \Delta\mathbf{w}\left(m\right)$$

9. Change in weights (hidden-output)

$$\Delta w_{kj} = \eta \delta_k y_j = \eta\left(t_k - z_k\right) f'\left(net_k\right) y_j$$

10. Derivation

$$f'\left(net\right) = \frac{2abe^{-bx}}{1+2e^{-bx}+e^{-2bx}}$$

11. Change in weights (input-hidden)

$$\Delta w_{ji} = \eta x_i \delta_j = \eta\left[\sum_{k=1}^{c} w_{kj}\delta_k\right] f'\left(net_j\right) x_i$$

Appendix B: Code Description

Our code consists of several files, other than the actual program itself [see Appendix D for codes]. There is the user input file (nnclassparam.dat), the program-specific header file (nntimers.h), the program used for the timings (timers.c), the portable batch system for the program (nnclass.pbs), and the makefile created to compile the program (Makefile). The actual program itself is nnclass.c [see Appendix C for code].

The user input data file nnclassparam.dat consists of the inputs that the user would have had to type in when prompted during the execution of the program. It has been placed in its own file with each input on a different line so that the code, if run in parallel, would not be prompted for these inputs by every processor involved in the parallelization.

The program-specific header file nntimers.h consists of declarations of functions that are used within the function timers.c and external declarations of the variables within timers.c so as to enable the usage of subroutines contained within timers.c, in our program, nnclass.c.

The C timing program timers.c consists of a series of functions to record the amount of time that is takes to perform a section of code with the function timer_start called at its start and timer_end at its completion. This is done by recording the time of day when either the function timer_start or timer_end is called. The time recorded by timer_start is then subtracted from the time recorded by timer_end thus recording the amount of time passed during the duration of the code encapsulated by calls to these two functions. In the program, most subroutines have been timed.

The portable batch system for our program, nnclass.pbs, was necessary to create so that our program could be run on blackbear. (On blackbear all jobs must be submitted to PBS. If they are not, they will be killed.) PBS is a networked subsystem for submitting, monitoring, and controlling a workload of batch jobs on one or more systems. Batch means that the job will be scheduled for execution at a time chosen by the subsystem according to a defined policy and the availability of resources. For a normal batch job such as this one, the standard output and standard error of the job will be returned to files available to the user when the job is complete. A typical job such as ours is a shell script and a set of attributes that provide resource and control information about the job.

The makefile for our program, Makefile, compiles our program for execution when "make" is typed. It makefile describes the relationships among files in our program, and states the commands for updating each file. In our program, as is typically, the executable file is updated from object files, which are in turn made by compiling source files. Before the makefile is run, timers.c must be compiled with the standard GNU C compiler GCC as follows "gcc –c timers.c" with the -c option so that GCC ignores any unrecognized input files (those that do not require compilation or assembly). This creates a ".o" file timers.o which will be used later by the makefile. The program itself is then compiled with the makefile as follows "gcc –c –g nnclass.c" using the "–g" option so that no optimization takes place and again using the "-c" option. This again creates a ".o" file this time nnclass.o which will be used later by the makefile. The makefile then links all object files and libraries, by doing the following "gcc -g -o nnclass nnclass.o timers.o –lm". It uses the "-o" option to link the object files nnclass.o and nntimers.o to the program executable nnclass, using the "-lm" option to include the standard math library in the link, and again using the "-g" option.

Now typing in six arguments can run the program. The first required argument is the name of the program executable (nnclass) and the second is the name of the user input data file (nnclassparam.dat). The third required argument is the name of the input data file that will be used to train the neural network (data_set_ALL_AML_train.txt) [see Appendix E]. The fourth required argument is the name of the data file that contains the desired output for both the input data file that will be used to train the neural network and the input data file that will be used to test the training of the neural network (table_ALL_AML_samples.txt) [see Appendix E]. The fifth argument required is the name of the input data file that will be used to test the training of the neural network (data_set_ALL_AML_independent.txt) [see Appendix E], and the sixth is desired name of the output file (nnclass.out).

When all six arguments have been entered in, the program gives the user the option to correct any default parameters incorrect for the particular data files that they choose to use. (Throughout the program, user inputs are obtained by opening the user input data file, nnclassparam.dat, and reading the inputs from there.) Then the program gives the user a choice of preprocessing scheme. They can decide to delete all of the values for gene expression that are identical for every patient and / or delete all of the values for gene expression for genes that are absent in every patient. They can also decide whether to use only the fifty genes selected by the MIT Whitehead group as input data, or all or the genes. If they choose to, they can decide not to do a pre-processing scheme. The program then makes the user choose the number of nodes in the hidden layer, jmax, which by default is set equal to the number of genes in the input data file, genes, multiplied by the number of patients, patients, divided by the quantity ten times genes plus ten times the number of nodes in the output layer, kmax, plus one (based on Duda et al. heuristic [2] p. 311).

$$\frac{genes + patients}{10 \times (genes + k\max)} + 1$$

The program then dynamically allocates memory for the arrays by using pointers to certain input variables as dimensions in the arrays and exits the program if there is not enough memory to allocate the arrays. It then opens the input data file and reads it into the arrays. Now that the memory for the dynamic arrays has been allocated, if the program is unable to open a data file, it deallocates the memory for the arrays before exiting the program so as not to unnecessarily take up hard-drive space after the program stops running. It reads in the gene accession numbers for each gene, the call (indication of gene absence – A, presence – P, or marginal – M) for each gene, and the gene expression values for each gene taking care to index these properly with the correct patient numbers. (The patient numbers are out of sequence in the file). The input file is then closed because it will not be needed for the rest of the program; also, leaving the file open slows computations. The training file is opened and the patient numbers are read in as well as subtype designations, which are converted from ALL to negative one and AML to positive one, and then stored an array, t. The training data file is closed because it will not be needed until the later in the program, and the pre-selected preprocessing functions are performed. The input layer then assigns the numerical gene intensity values read in from the input file, to the input for the neural network, x. It does so according to which preprocessing scheme was chosen, namely whether all 7129 genes are to be used as input or just the fifty genes. Now the user chooses the type of training to use, batch or stochastic.

Here the program begins its first iteration. The patients will be feed into the neural network one by one but the way that this is done depends on the type of training being used. If the training for the neural network is stochastic, a random patient is chosen to be put through the neural network until the number of random patients chosen is equivalent to the number of patients (randomly chosen patients may be picked more than once). If the training is batch, the patients are sequentially analyzed.

Now the hidden layer, calculates the value of the net activation, the weighted sum of its inputs, for the hidden layer, netj, which is the sum of all of the values of gene expression for a given patient, x, multiplied by their weights, wji [see equation 1 in Appendix A]. It also calculates the value of the net activation function, fnetj [see equation 5 in Appendix A], which is equivalent to the output of the hidden layer, y [see equation 2 in Appendix A].

Now in the output layer, the value of the net activation for the output layer, netk, which is the sum of all of the outputs from the hidden layer, y, multiplied by their weights, wkj, is calculated [see equation 3 in Appendix A]. It also calculates the value of the net activation function, fnetj [see equation 5 in Appendix A], which is equivalent to the output from the output layer, z [see equation 4 in Appendix A]. Now the compounded least squares error, Jnew, is calculated. This is done by adding the previous least squares error calculated, Jold, to one half of the square of the difference between the value of the training data, t, (the desired results) and the output from the output layer, z [see equation 6 in Appendix A].

Here backpropagation calculates the derivative of the net activation function for the hidden and output layers, fPRIMEnetj and fPRIMEnetk respectively [see equation 10 in Appendix A]. It also calculates the change in the weights from the input to hidden and hidden to output layers, DELTAwji [see equation 11 in appendix A] and DELTAwkj [see equation 9 in Appendix A] respectively. It also calculates the change in the output k, DELTAk [see equation 7 in Appendix A].

Where the weights are updated depends on the user's choice of training method. If the training being used is stochastic, the weights are updated now by calculating the new weights for the input to hidden and hidden to output layers [see equation 8 in Appendix A]. If the training being used is batch, the weights are updated when the number of random patients chosen equals the total number of patients, in other words, after each iteration. Now the change in the least squares error, DELTAJ, is calculated by taking the absolute value of the current least squares error minus the previous least squares error.

$$\Delta J = \left| J(m) + J(m+1) \right|$$

Then the DELTAJ is tested to see whether or not it is less than the stopping criterion threshold value, THETA. If it is, then the current iteration becomes the last one and if it is not, then a new iteration starts until it either becomes so or the number of iterations reaches one hundred.
Jold is then set equal to Jnew so that Jnew may be reassigned during the next iteration without loosing the previous J.

At this point in the program, all of the iterations have been completed and the training data is equivalent to the output from the neural net. With the training done, the neural network may now be tested to see how well it was actually trained. The testing data file is opened and read into arrays exactly the same way as the input data file was taking care to index the input properly with the correct patient numbers (the patient numbers are out of sequence in the file). It reads the gene accession numbers for each gene, the call (indication of gene absence – A, presence – P, or marginal – M) for each gene, and the gene expression values for each gene. The testing file is then closed; it will not be needed for the rest of the program. The gene intensity values for each gene are then passed through the input layer where they are assigned to x depending upon how many genes are used as input, 50 or 7129. Then x is passed to the hidden and output layers after which the testing of how well the neural network has been trained is done. If new weights yielded from the training neural network are correct, now the numerical output of the neural network, z, should be equivalent to the actual classifications of the patients in the testing data set with negative one corresponding to ALL and positive one corresponding to AML.

Now with the testing done, the output data file is opened and the patient numbers and subtype designation for each patient are printed to it. Several other parameters, such as the number of genes used as input, the type of training used, the learning rate, and the number of nodes in the hidden layer, are printed as well. The output file is then closed and memory is deallocated for the dynamic arrays so that hard-drive space is not to unnecessarily taken up after the program stops running. After this is done, the program is over and main is closed.

Appendix C: Code

nnclass.c

```
/* Manual Compilation */
/* gcc -c nnclass.c                      (compiles nnclass.c to nnclass.o) */
/* gcc -c timers.c                       (compiles timers.c to timers.o) */
/* gcc -o nnclass nnclass.o timers.o -lm (links all object files & libraries)
*/

/* Automated Compilation */
/* make */
/* (gcc -c nnclass.c) - (compiles nnclass.c to nnclass.o) */
/* (gcc -c timers.c) - (compiles timers.c to timers.o) */
/* (gcc -o nnclass nnclass.o timers.o -lm) - (links all object files &
libraries) */

/* Execution */
/* nnclass nnclassparam.dat data_set_ALL_AML_train.txt
data_set_ALL_AML_independent.txt table_ALL_AML_samples.txt nnclass.out */

/* Remove .o files */
/* make clean */

/* Debugging */
/* gcc -c nnclass.c (compiles nnclass.c to nnclass.o) */
/* gcc -c timers.c  (compiles timers.c to timers.o) */
/* gcc -o nnclass nnclass.o timers.o -lm -g (links all object files &
libraries for debugger) */
/* mv nnclass run */
/* cd run */
/* gdb nnclass */
/* nnclass nnclassparam.dat data_set_ALL_AML_train.txt
data_set_ALL_AML_independent.txt table_ALL_AML_samples.txt nnclass.out */
/* (This error message will appear - Undefined command: "nnclass".  Try
"help".) */
/* run nnclass nnclassparam.dat data_set_ALL_AML_train.txt
data_set_ALL_AML_independent.txt table_ALL_AML_samples.txt nnclass.out */

/* Printing */
/* enscript -2rGE nnclass.c -o file.ps */
/* lpr -Pnetprt1 file.ps (prints at front desk) */

/* Parallel job submission */
/* qsub -o nnclass.out -e nnclass.log nnclass.pbs */

/* C Libraries */
#include<stdio.h>  /* standard C library */
#include<math.h>   /* standard math library */
#include<stdlib.h> /* included for exit (quit program), and rand (random
number generator) */
#include<string.h> /* included for strcmp (string compare), and strchr
(string search for a character) */

#ifdef mpi         /* if using mpi */
/* standard mpi (message passing interface) library */
#include "mpi.h"
#endif
```

```
/* program-specific includes */
#include "nntimers.h"

/* Files */
FILE *fp;                                          /* pointer to selected
file */
char paramfile[] = "nnclassparam.dat";             /* default file of
input parameters set by user (for parallel program) */
char infile[] = "data_set_ALL_AML_train.txt";      /* default input data
file used to train the neural network */
char testfile[] = "data_set_ALL_AML_independent.txt"; /* default input data
file used to test the training of the neural network
*/
char outcomefile[] = "table_ALL_AML_samples.txt";     /* default outcome data
file (both training and testing data sets) */
char outfile[] = "nnclass.out";                    /* default output file
*/

/* Declaration/initialization of variables */
int whoami = 0;          /* processor id; default value is 0 for serial runs
*/
int nproc = 1;           /* number of processors */
int d = 0;               /* number of nodes in input layer, each one
corresponding to an input */
int genes = 0;           /* number of genes in the input or testing data
files */
int totalgenes = 0;      /* number of genes in the input and testing data
files */
int patients = 0;        /* larger number of patients in the input and
training data sets */
int npat = 0;            /* inputpatients if training, testingpatients if
testing */
int inputpatients = 0;   /* number of patients in input data set */
int testingpatients = 0; /* number of patients in testing sets */
int totalpatients = 0;   /* total number of patients in input and testing
data sets */
int inputcolumns = 0;    /* number of columns in input data file */
int trainingcolumns = 0; /* number of columns in training data file */
int testingcolumns = 0;  /* number of columns in testing data file */
int inputrows = 0;       /* number of rows in input data file */
int trainingrows = 0;    /* number of rows in training data file */
int testingrows = 0;     /* number of rows in testing data file */
int blankint = 0;        /* reassigned to read in useless integers */
/* double blankdouble1 = 0.0;/* reassigned to read in useless long floating
point values */
/* double blankdouble2 = 0.0;/* reassigned to read in useless long floating
point values */
/* double blankdouble3 = 0.0;/* reassigned to read in useless long floating
point values */
double mu1 = 0.0;        /* mean gene expression value for patient 1 of
training set */
int letters = 0;         /* maximum number of letters used in input */
char blankchar[250];     /* reassigned to read in useless chars */
int preprocessing = 0;   /* for preprocessing scheme */
char trainingtype[11];   /* user input variable for the type of training
that will be used for the neural network */
```

```
int input = 0;            /* user input for verification of correct variables
*/
int epoch = 0;            /* the counter for the epoch/interation */
char *trainortest;        /* designates if training or testing the neural
network */

/* Constants  */
#define INFINITY 100          /* used for all intents and purposes as
infinity */
int randompatients[INFINITY]; /* running record of the patients used and in
what order */

/* Declaration of Indices */
int i = 0;      /* input layer index */
int j = 0;      /* hidden layer(s) index */
int k = 0;      /* output layer index */
int p = 0;      /* patients index */
int g = 0;      /* genes index */
int s = 0;      /* c and timers index */
int cn = 0;     /* columns index */
int r = 0;      /* rows index */
int l = 0;      /* letters index */
int q = 0;      /* assigned to realpatients[p] */
int n = 0;      /* pstr index */
int pat = 0;    /* patients index for iteration/epoch */
/* int ranpat = 0;  random patients index */

/* Initialization of Dynamic Arrays */
int *realpatients;            /* patient numbers in input */
char *fiftygenes[50];         /* 50 genes selected by MIT Whitehead group */
char **Gene_Accession_Number; /* Gene Accession Number in input */
double **array;               /* gene intensity values */
double **x;                   /* output from input layer, one corresponding
to each gene */
char **call;                  /* call[genes][patients]; */
double **y;                   /* output from hidden layer */
double ***wji;                /* weights for output from input layer */
double **netj;                /* net activation for output from input layer
*/
double **fnetj;               /* equation */
double **z;                   /*  */
double ***wkj;                /* weights for output from hidden layer */
double **netk;                /* net activation for output from hidden layer
*/
double **fnetk;               /* equation */
double **t;                   /*  */
double **fPRIMEnetj;          /* derivitive netj */
double **DELTAwji;            /* change in wji */
double **fPRIMEnetk;          /* derivitive of netk */
double **DELTAwkj;            /* change in wkj */
double **DELTAk;              /* change in k */

/* Initialization of Preprocessing2 Variables */
int Acounter = 0;
int Pcounter = 0;
int Mcounter = 0;
```

```
/* Initialization of Input_Layer Variables */
char line[5120];
char *pstr;

/* Initialization of Hidden_Layer Variables */
int jmax = 0;   /* number of nodes in hidden layer */
double a = 0.0; /* fnetj variable */
double b = 0.0; /* fnetj variable */

/* Initialization of Output_Layer Variables */
int kmax = 0; /* number of nodes in output layer */
int nH = 0;   /* number of hidden layers */

/* Initialization of Back_Propagation Variables */
int c = 0;           /* length of target and network output
vectors; number of subtypes */
int m = 0;           /* iteration */
double ETA = 0.0;    /* learning rate */
double THETA = 0.0;  /* stopping criterion threshold value */
double Jold = 0.0;   /* old least squares error */
double Jnew = 0.0;   /* new least squares error */
double DELTAJ = 0.0; /* change in least squares error */

/* Function Declarations */
void allocate_dynamic_array_memory();
void input_layer(FILE* fp);
void compute_baseline(); /* stores gene expression b\values for baseline
patient, training patient #1, and evaluates mean-value, mu1 */
void rescale(); /* mean-centers with respect to baseline, and normalizes with
respect to standard deviation; rescales the data so that all values do not
turn into zero when functions are taken to a negative power of some variation
of the input */
void preprocessing1(); /* deletes all of the values for gene expression that
are identical for every patient */
void preprocessing2(); /* deletes all of the values for gene expression for
genes that are absent in every patient */
void preprocessing3(); /* uses only the fifty genes selected by the MIT
Whitehead group as input data */
void hidden_layer(); /* */
void output_layer(); /* */
void backpropagation(); /* */
void update_weights(); /* */
void print_subtypes(FILE* fp); /* specialized subroutine for 2 class AML/ALL
problem */
void deallocate_dynamic_array_memory(); /* */
void terminate(); /* */

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

#ifdef mpi
   /* parallel initialization */
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD,&whoami);
   MPI_Comm_size(MPI_COMM_WORLD,&nproc);
#endif

   /* int argc = number of arguments */
   /* char* argv[5] = pointer to array contatining arguments */
   /* argv[0] = executable program name */
   /* argv[1] = parameters file                 (default: paramfile) */
   /* argv[2] = input data file used for training (default: infile)    */
   /* argv[3] = input data file used for testing  (default: testfile)  */
   /* argv[4] = training data file              (default: outcomefile) */
   /* argv[5] = output file                     (default: outfile)   */

   /* %s = char */
   /* %d = int */
   /* %f = float */

   /* set defaults */
   genes = 7129;
   totalgenes = 2 * genes;
   inputpatients = 38;
   testingpatients = 34;
   totalpatients = inputpatients + testingpatients;
   letters = 30;
   inputrows = genes + 1;
   trainingrows = totalpatients + 19; /* 20 */
   testingrows = inputrows;
   inputcolumns = 2 * inputpatients + 2;
   trainingcolumns = 10;
   testingcolumns = 2 * testingpatients + 2;
   preprocessing = -1;
   a = 1.716; /* static */
   b = 2.0/3.0; /* static */
   c = 2; /* -1 = ALL; 1 = AML */
   kmax = 1; /* 1 output node (special case for 2 subtype classification) */
   nH = 1;
   ETA = 0.1; /* suggested starting learning rate, Duda et al. p. 313 */
   THETA = 0.001;
   m = 1;

   if (argc != 6) {
      /* argc number of arguments */
      /* argv array contatining arguments */
      if (whoami == 0) {
         printf("Usage: <program executable> <parameters file> <input data
file> <test data file> <training data file> <output data file>.\n");
         printf("Program Executable: nnclass\n");
         printf("Default Parameters:\n");
         printf("Parameters File:    %s\n", paramfile);
         printf("Input Data File:    %s\n", infile);
         printf("Test Data File:     %s\n", testfile);
         printf("Training Data File: %s\n", outcomefile);
         printf("Output File:        %s\n\n", outfile);
      } /* if */
#ifdef mpi
      MPI_Finalize();
#endif
      exit(0);
   } /* if */
   else {
      if (whoami == 0) {
         printf("Program:            nnclass.c\n");
         printf("Program Executable: %s\n", argv[0]);
         printf("Parameters File:    %s\n", argv[1]);
         printf("Input Data File:    %s\n", argv[2]);
         printf("Test Data File:     %s\n", argv[3]);
         printf("Training Data File: %s\n", argv[4]);
         printf("Output File:        %s\n\n", argv[5]);
      } /* if */
   } /* else */

   /* clear all timers (can use up to 20) */
   for (s = 0; s < 13; s++) {
      timer_clear(s);
   } /* for */

   timer_start(0); /* timer for entire program */
   timer_start(1); /* timer for user interface */
   while (input < 1 || input > 2) {
      if (whoami == 0) {
         printf("Default Parameters:\n");
         printf("Number of Patients in the Input Data File:            %d\n",
inputpatients);
         printf("Number of Patients in the Testing Data File:          %d\n",
testingpatients);
         printf("Total Number of Patients:                             %d\n",
totalpatients);
         printf("Number of Genes in the Input or Testing Data Files:  %d\n",
genes);
         printf("Number of Genes in the Input and Testing Data Files: %d\n",
totalgenes);
         printf("Number of Columns in the Input Data File:            %d\n",
inputcolumns);
         printf("Number of Columns in the Testing Data File:          %d\n",
testingcolumns);
         printf("Number of Columns in the Training Data File:         %d\n",
trainingcolumns);
         printf("Number of Rows in the Input Data File:               %d\n",
inputrows);
         printf("Number of Rows in the Testing Data File:             %d\n",
testingrows);
         printf("Number of Rows in the Training Data File:            %d\n",
trainingrows);
         printf("Maximum Number of Letters in any String:             %d\n",
letters);
         printf("Number of Subtypes:                                  %d\n",
c);
         printf("Number of Neurons in the Output Layer:               %d\n",
kmax);
         printf("Number of Hidden Layers:                             %d\n",
nH);
         printf("Learning Rate:                                       %lf\n",
ETA);
         printf("Stopping Criterion Threshold Value:                  %lf\n",
THETA);
         printf("Type '1' if the default parameters are correct.\n");
         printf("Type '2' to make corrections. ");
      } /* if */
      fp = fopen(argv[1], "rb"); /* open parameters file */
      if (fp == NULL) { /* if unable to open parameters file, */
         if (whoami ==0) {
            printf("Unable to open input data file, %s.\n", argv[1]);
#ifdef mpi
            MPI_Finalize();
#endif
            exit(0); /* terminate program */
         } /* if */
      } /* if */
      else {
         /* read number of data lines specified by file */
         fgets(line,5120,fp); /* while not end of file, */
         sscanf(line, "%d", &input);
      } /* else */
   } /* while */

   while (input == 1) {
      input = 0;
   } /* while */

   while (input == 2) {
      if (whoami == 0) {
         printf("Please enter the number of genes in the input or testing
data files.\n");
         printf("The number of genes in the testing data file must be
equivalent to the number of genes in the input data file.\n");
         printf("Type '0' to use the default number of genes in the input or
testing data files %d.\n", genes);
      } /* if */
      fgets(line,5120,fp); /* while not end of file, */
      sscanf(line, "%d", &blankint);
      if (blankint > 0) {
         genes = blankint;
         input = 2;
      } /* if */
      else if (blankint < 0) {
         input = 1;
      } /* else if */
      totalgenes = 2 * genes;
      if (whoami == 0) {
         printf("Please enter the number of patients in the input data
file.\n");
         printf("Type '0' to use the default number of patients in the input
data file %d.\n", inputpatients);
      } /* if */
      fgets(line,5120,fp); /* while not end of file, */
      sscanf(line, "%d", &blankint);
      if (blankint > 0) {
         inputpatients = blankint;
      } /* if */
      else if (blankint < 0) {
         input = 1;
      } /* else if */
      if (whoami == 0) {
         printf("Please enter the number of patients in the testing data
file.\n");
```

```
         printf("Type '0' to use the default number of patients in the
testing data file %d.\n", testingpatients);
      } /* if */
      fgets(line,5120,fp); /* while not end of file, */
      sscanf(line, "%d", &blankint);
      if (blankint > 0) {
         testingpatients = blankint;
      } /* if */
      else if (blankint < 0) {
         input = 1;
      } /* else if */
      totalpatients = inputpatients + testingpatients;
      if (whoami == 0) {
         printf("Please enter the input number of patients in the input and
testing data file.\n");
         printf("Type '0' to use the default total number of patients in the
input and testing data files %d.\n", totalpatients);
      } /* if */
      fgets(line,5120,fp); /* while not end of file, */
      sscanf(line, "%d", &blankint);
      if (blankint > 0) {
         totalpatients = blankint;
      } /* if */
      else if (blankint < 0) {
         input = 1;
      } /* else if */

      if (whoami == 0) {
         printf("Please enter the number of rows in the input data file.\n");
         printf("Type '0' to use the default number of rows in the input data
file %d.\n", inputrows);
      } /* if */
      fgets(line,5120,fp); /* while not end of file, */
      sscanf(line, "%d", &blankint);
      if (blankint > 0) {
         inputrows = blankint;
      } /* if */
      else if (blankint < 0) {
         input = 1;
      } /* else if */
      if (whoami == 0) {
         printf("Please enter the number of columns in the input data
file.\n");
         printf("Type '0' to use the default number of columns in the input
data file %d.\n", inputcolumns);
      } /* if */
      fgets(line,5120,fp); /* while not end of file, */
      sscanf(line, "%d", &blankint);
      if (blankint > 0) {
         inputcolumns = blankint;
      } /* if */
      else if (blankint < 0) {
         input = 1;
      } /* else if */
      if (whoami == 0) {
         printf("Please enter the number of rows in the training data
file.\n");
```

```
         printf("Type '0' to use the default number of rows in the training
data file %d.\n", inputrows);
      } /* if */
      fgets(line,5120,fp); /* while not end of file, */
      sscanf(line, "%d", &blankint);
      if (blankint > 0) {
         trainingrows = blankint;
      } /* if */
      else if (blankint < 0) {
         input = 1;
      } /* else if */
      if (whoami == 0) {
         printf("Please enter the number of columns in the training data
file.\n");
         printf("Type '0' to use the default number of columns in the
training data file %d.\n", trainingcolumns);
      } /* if */
      fgets(line,5120,fp); /* while not end of file, */
      sscanf(line, "%d", &blankint);
      if (blankint > 0) {
         trainingcolumns = blankint;
      } /* if */
      else if (blankint < 0) {
         input = 1;
      } /* else if */
      if (whoami == 0) {
         printf("Please enter the maximum number of letters in a string.\n");
         printf("Type '0' to use the default maximum number of letters in a
string %d.\n", trainingcolumns);
      } /* if */
      fgets(line,5120,fp); /* while not end of file, */
      sscanf(line, "%d", &blankint);
      if (blankint > 0) {
         trainingcolumns = blankint;
      } /* if */
      else if (blankint < 0) {
         input = 1;
      } /* else if */
      if (whoami == 0) {
         printf("Please enter the number of subtypes.\n");
         printf("Type '0' to use the default number of subtypes %d.\n", c);
      } /* if */
      fgets(line,5120,fp); /* while not end of file, */
      sscanf(line, "%d", &blankint);
      if (blankint > 0) {
         c = blankint;
      } /* if */
      else if (blankint < 0) {
         input = 1;
      } /* else if */
      if (whoami == 0) {
         printf("Please enter the number of nodes in the output layer.\n");
         printf("Type '0' to use the default number of nodes in the output
layer %d.\n", kmax);
      } /* if */
      fgets(line,5120,fp); /* while not end of file, */
      sscanf(line, "%d", &blankint);
      if (blankint > 0) {
         kmax = blankint;
      } /* if */
      else if (blankint < 0) {
         input = 1;
      } /* else if */
      if (whoami == 0) {
         printf("Please enter the number of hidden layers.\n");
         printf("Type '0' to use the default number of hidden layers %d.\n",
nH);
      } /* if */
      fgets(line,5120,fp); /* while not end of file, */
      sscanf(line, "%d", &blankint);
      if (blankint > 0) {
         nH = blankint;
      } /* if */
      else if (blankint < 0) {
         input = 1;
      } /* else if */
      if (whoami == 0) {
         printf("Please enter the learning rate.\n");
         printf("Type '0' to use the default learning rate %lf.\n", ETA);
      } /* if */
      fgets(line,5120,fp); /* while not end of file, */
      sscanf(line, "%d", &blankint);
      if (blankint > 0) {
         ETA = blankint;
      } /* if */
      else if (blankint < 0) {
         input = 1;
      } /* else if */
      if (whoami == 0) {
         printf("Please enter the stopping criterion threshold value.\n");
         printf("Type '0' to use the default stopping criterion threshold
value %lf.\n", THETA);
      } /* if */
      fgets(line,5120,fp); /* while not end of file, */
      sscanf(line, "%d", &blankint);
      if (blankint > 0) {
         THETA = blankint;
      } /* if */
      else if (blankint < 0) {
         input = 1;
      } /* else if */
      if (whoami == 0) {
         printf("Adjusted Parameters:\n");
         printf("Number of Patients in the Input Data File:           %d\n",
inputpatients);
         printf("Number of Patients in the Testing Data File:         %d\n",
testingpatients);
         printf("Total Number of Patients:                            %d\n",
totalpatients);
         printf("Number of Genes in the Input or Testing Data Files:  %d\n",
genes);
         printf("Number of Genes in the Input and Testing Data Files: %d\n",
totalgenes);
```

```
         printf("Number of Columns in the Input Data File:            %d\n",
inputcolumns);
         printf("Number of Columns in the Testing Data File:          %d\n",
testingcolumns);
         printf("Number of Columns in the Training Data File:         %d\n",
trainingcolumns);
         printf("Number of Rows in the Input Data File:               %d\n",
inputrows);
         printf("Number of Rows in the Testing Data File:             %d\n",
testingrows);
         printf("Number of Rows in the Training Data File:            %d\n",
trainingrows);
         printf("Maximum Number of Letters in any String:             %d\n",
letters);
         printf("Number of Subtypes:                                  %d\n",
c);
         printf("Number of Neurons in the Output Layer:               %d\n",
kmax);
         printf("Number of Hidden Layers:                             %d\n",
nH);
         printf("Learning Rate:                                       %lf\n",
ETA);
         printf("Stopping Criterion Threshold Value:                  %lf\n",
THETA);
      } /* if */
      if (input == 2) {
         if (whoami == 0) {
            printf("Type '1' if correct.\n");
            printf("Type '2' to make corrections. ");
         } /* if */
         fgets(line,5120,fp); /* while not end of file, */
         sscanf(line, "%d", &blankint);
      } /* if */
      else if (input == 1) {
         input = 2;
      } /* else if */
      while (input <= 0 || input > 2) {
         if (whoami == 0) {
            printf("Type '1' if correct.\n");
            printf("Type '2' to make corrections. ");
         } /* if */
         fgets(line,5120,fp); /* while not end of file, */
         sscanf(line, "%d", &blankint);
      } /* while */
   } /* while */

   if (inputpatients > testingpatients) {
      patients = inputpatients;
   } /* if */
   else { /* if (testingpatients > inputpatients) { */
      patients = testingpatients;
   } /* else */

   while (preprocessing < 0 || preprocessing > 6) {
      if (whoami == 0) {
         printf("Please enter the type of pre-processing that you would like
to use.\n");
```

```
         printf("If you would like to delete all of the values for gene
expression that are identical for every patient, type '1'.\n");
         printf("If you would like to delete all of the values for gene
expression for genes that are absent in every patient, type '2'.\n");
         printf("If you would like to do both, type '3'.\n");
         printf("If you would like to use only the fifty genes selected by
the MIT Whitehead group as input data, type '4'.\n");
         printf("If you would not like to use a pre-processing scheme, type
'5'.\n");
         printf("Type '0' for default. ");
      } /* if */
      fgets(line,5120,fp); /* while not end of file, */
      sscanf(line, "%d", &preprocessing);
      if (whoami == 0) {
         printf("%d\n", preprocessing);
      } /* if */
      if (preprocessing == 1 || preprocessing == 2 || preprocessing == 3 ||
preprocessing == 5) {
         d = genes;
      } /* if */
      else if (preprocessing == 4 || preprocessing == 0) {
         d = 50;
      } /* else if */
   } /* while */

   jmax = (genes * inputpatients) / (10 * (genes + kmax)) + 1; /* based on
Duda et al. heuristic, p. 311 */
   input = 1;
   while (input == 1) {
      if (whoami == 0) {
         printf("Please enter the number of nodes in the hidden layer.\n");
         printf("Type '0' to use the default number of nodes in the hidden
layer %d. \n", jmax);
      } /* if */
      fgets(line,5120,fp); /* while not end of file, */
      sscanf(line, "%d", &blankint);
      if (blankint > 0) {
         jmax = blankint;
         input = 0;
      } /* if */
      else if (blankint < 0) {
         input = 1;
      } /* else if */
      else {
         input = 0;
      } /* else */
   } /* while */
   timer_stop(1);

   trainortest = "train";
   npat = inputpatients;

   fclose(fp); /* close parameter file */

   timer_start(2);
   allocate_dynamic_array_memory(); /* call the function
allocate_dynamic_array_memory */
```

```
timer_stop(2);

timer_start(3);
fp = fopen(argv[2], "rb"); /* open input data file */
if (fp == NULL) { /* if unable to open input data file, */
   if (whoami == 0) {
      printf("Unable to open input data file, %s.\n", argv[2]);
   } /* if */
   terminate(); /* terminate program */
} /* if */
else {
   /* read number of data lines specified by the file */
   for (r = 0; r < inputrows; r++) {
      fgets(line,5120,fp); /* while not end of file, */
      pstr = line;
      if (r == 0) {
         for (n = 0; n < 2; n++) {
            pstr = strchr(pstr, '\t');
            pstr++;
         } /* for */
         for (p = 0; p < inputpatients; p++) {
            sscanf(pstr, "%d", &realpatients[p]);
            realpatients[p] = realpatients[p] - 1;
            for (n = 0; n < 2; n++) {
               pstr = strchr(pstr, '\t');
               pstr++;
            } /* for */
         } /* for */
      } /* if */
      else {
         g = r - 1;
         pstr = strchr(pstr, '\t');
         pstr++;
         sscanf(pstr, "%s", Gene_Accession_Number[g]);
         for (p = 0; p < inputpatients; p++) {
            pstr = strchr(pstr, '\t');
            pstr++;
            /* assign values to array */
            q = realpatients[p];
            sscanf(pstr, "%lf", &array[g][q]);
            /* assign values to call */
            pstr = strchr(pstr, '\t');
            pstr++;
            /* assign values to call */
            sscanf (pstr, "%c", &call[g][q]);
         } /* for */
      } /* for */
   } /* else */
   fclose(fp); /* close the input data file */
} /* else */
timer_stop(3);

timer_start(4);
fp = fopen(argv[4], "rb"); /* open training data file */
if (fp == NULL) { /* if unable to open training data file */
   if (whoami == 0) {
      printf("Unable to open training data file, %s.\n", argv[4]);
      } /* if */
      terminate(); /* terminate the program */
   } /* if */
   else {
      /* read the number of data lines specified by the file */
      for (r = 0; r < trainingrows; r++) {
         fgets(line,5120,fp); /* while not end of file, */
         if (r > 18) {
            pstr = line;
            sscanf(pstr, "%d", &q);
            q = q - 1;
            pstr = strchr(pstr, 'A');
            pstr++;
            sscanf(pstr, "%s", &blankchar);
            if (!strcmp (blankchar, "LL")) {
               t[0][q] = -1;
            } /* if */
            else { /* if (!strcmp, blankchar "ML") */
               t[0][q] = 1;
            } /* else */
            fgets(line,5120,fp);
            if (q == 37) {
               for (i = 0; i < 6; i++) {
                  fgets(line,5120,fp);
               } /* for */
            } /* if */
         } /* if */
      } /* for */
      fclose(fp); /* close training data file */
   } /* else */
   timer_stop(4);

   if (preprocessing == 4 || preprocessing == 0) {
      timer_start(5);
      preprocessing3(); /* call the function preprocessing3 */
      timer_stop(5);
   } /* else if */
   timer_start(6);
   input_layer(fp); /* call the function input_layer */
   timer_stop(6);
   if (preprocessing == 1 || preprocessing == 3) {
      timer_start(5);
      preprocessing1(); /* call the function preprocessing1 */
      timer_stop(5);
   } /* if */
   else if (preprocessing == 2) {
      timer_start(5);
      preprocessing2(); /* call the function preprocessing2 */
      timer_stop(5);
   } /* else if */

   input = 1;
   while (input == 1) {
      if (whoami == 0) {
         printf("Which type of training would you like to use for the neural
network, batch or stochastic? ");
      } /* if */
      fp = fopen(argv[1], "rb"); /* open parameters file */
      if (fp == NULL) { /* if unable to open parameters file, */
         if (whoami == 0) {
            printf("Unable to open input data file, %s.\n", argv[1]);
            terminate(); /* terminate program */
         } /* if */
      } /* if */
      else {
         /* read the number of data lines specified by the file */
         for (g = 0; g < 4; g++) {
            fgets(line,5120,fp); /* while not end of file, */
         } /* for */
         sscanf(line, "%s", &trainingtype);
         if (whoami == 0) {
            printf("%s\n", trainingtype);
         } /* if */
      } /* else */
      if (!strcmp (trainingtype, "batch") || !strcmp (trainingtype,
"stochastic")) {
         input = 0;
      } /* if */
   } /* while */

  /******************* START TRAINING SECTION *******************/

   for (m = 0; m < INFINITY; m++) {
      Jnew = 0.0;
      p = -1;
      for (pat = 0; pat < inputpatients; pat++) {
         if (!strcmp (trainingtype, "stochastic")) {
            p = (int) ((float) inputpatients * rand() / (RAND_MAX + 1.0));
            randompatients[pat] = p;
         } /* if */
         else { /* if (!strcmp (trainingtype, "batch")) */
            p++;
         } /* else */

         /* Feed Forward */
         timer_start(7);
         hidden_layer(); /* call the function hidden_layer */
         timer_stop(7);

         timer_start(8);
         output_layer(); /* call the function output_layer */
         timer_stop(8);

         /* calculate J(w) */
         /* J(w) = 1/2 c_SUMMATION_k=1 (tk - zk)^2  (4) */
         for (k = 0; k < kmax; k++) {
            Jnew = Jnew + .5 * (pow((t[k][p] - z[k][p]), 2));
         } /* for */

         timer_start(9);
         backpropagation(); /* call the function backpropagation */
         timer_stop(9);

         if (!strcmp (trainingtype, "stochastic")) {
            timer_start(10);
            update_weights();
            timer_stop(10);
         } /* if */
      } /* for */
      if (!strcmp (trainingtype, "batch")) {
         timer_start(10);
         update_weights();
         timer_stop(10);
      } /* if */
      epoch++;
      /* calculate DELTAJ */
      /* DELTAJ = |J(m) - J(m-1)| */
      DELTAJ = fabs(Jnew - Jold); /* blankint); */
      if (DELTAJ < THETA) {
         break;
      } /* if */
      Jold = Jnew;
   } /* for */

   /******************* END TRAINING SECTION *******************/

   trainortest = "test";
   npat = testingpatients;
   timer_start(11);
   fp = fopen(argv[3], "rb"); /* open testing data file */
   if (fp == NULL) { /* if unable to open testing data file, */
      if (whoami == 0) {
         printf("Unable to open testing data file, %s.\n", argv[3]);
      } /* if */
      terminate(); /* terminate program */
   } /* if */
   else {
      /* read the number of data lines specified by the file */
      for (r = 0; r < testingrows; r++) {
         fgets(line,5120,fp); /* while not end of file, */
         pstr = line;
         if (r == 0) {
            for (n = 0; n < 2; n++) {
               pstr = strchr(pstr, '\t');
               pstr++;
            } /* for */
            for (p = 0; p < testingpatients; p++) {
               sscanf(pstr, "%d", &realpatients[p]);
               realpatients[p] = realpatients[p] - 1;
               for (n = 0; n < 2; n++) {
                  pstr = strchr(pstr, '\t');
                  pstr++;
               } /* for */
            } /* for */
         } /* if */
         else {
            g = r - 1;
            cn = genes + g;
            pstr = strchr(pstr, '\t');
            pstr++;
            sscanf(pstr, "%s", Gene_Accession_Number[cn]);
            input = 1;
            for (j = 0; j < genes; j++) {
               if (!strcmp (Gene_Accession_Number[j],
Gene_Accession_Number[cn])) {
                   input = 2;
               } /* if */
            } /* for */
            if (input == 1) {
               printf("The genes in the input data file and the testing data
file are not identical.\n");
               terminate(); /* terminate program */
            } /* if */
            for (p = 0; p < testingpatients; p++) {
               pstr = strchr(pstr, '\t');
               pstr++;
               /* assign values to array */
               q = realpatients[p] - inputpatients;
               sscanf(pstr, "%lf", &array[g][q]);
               /* assign values to call */
               pstr = strchr(pstr, '\t');
               pstr++;
               /* assign values to call */
               sscanf (pstr, "%c", &call[g][q]);
            } /* for */
         } /* for */
      } /* else */
      fclose(fp); /* close the testing data file */
   } /* else */
   timer_stop(11);

   timer_start(13);
   input_layer(fp); /* call the function input_layer */
   timer_stop(13);

   if (preprocessing == 1 || preprocessing == 3) {
      timer_start(12);
      preprocessing1(); /* call the function preprocessing1 */
      timer_stop(12);
   } /* if */
   else if (preprocessing == 2) {
      timer_start(12);
      preprocessing2(); /* call the function preprocessing2 */
      timer_stop(12);
   } /* else if */

   /* Test the Training of the Neural Network */
   p = -1;
   for (pat = 0; pat < testingpatients; pat++) {
      if (!strcmp (trainingtype, "stochastic")) {
         p = (int) ((float) testingpatients * rand() / (RAND_MAX + 1.0));
         randompatients[pat] = p;
      } /* if */
      else { /* if (!strcmp (trainingtype, "batch")) */
         p++;
      } /* else */

      /* Feed Forward */
      timer_start(14);
      hidden_layer(); /* call the function hidden_layer */
      timer_stop(14);

      timer_start(15);
      output_layer(); /* call the function output_layer */
      timer_stop(15);
   } /* for */

   timer_start(16);

   fp = fopen(argv[5], "w"); /* open output file */
   if (fp == NULL) { /* if unable to open output file */
      if (whoami == 0) {
         printf("Unable to open output file, %s.\n", argv[5]);
      } /* if */
      terminate(); /* terminate program */
   } /* if */
   else {
      if (whoami == 0) {
         printf("output data file opened\n");
      } /* if */
   } /* else */

   print_subtypes(fp); /* call the function print_subtypes */

   fclose(fp); /* close output data file */

   timer_stop(16);

   timer_start(17);
   deallocate_dynamic_array_memory();
   timer_stop(17);

   timer_stop(0);
   /* timings */
   /* timer(0) = program */
   /* timer(1) = user interface */
   /* timer(2) = allocation of dynamic array memory */
   /* timer(3) = input data file */
   /* timer(4) = training data file */
   /* timer(5) = preprocessing */
   /* timer(6) = training input layer */
   /* timer(7) = training hidden layer */
   /* timer(8) = training output layer */
   /* timer(9) = backpropagation */
   /* timer(10) = update weights */
   /* timer(11) = testing data file */
   /* timer(12) = testing preprocessing */
   /* timer(13) = testing input layer */
   /* timer(14) = testing hidden layer */
   /* timer(15) = testing output layer */
   /* timer(16) = print subtypes */
   /* timer(17) = deallocation of dynamic array memory */

#ifdef mpi
   MPI_Finalize();
```

```
#endif

   return 0;

} /* main */


void allocate_dynamic_array_memory() {
   /* array[a][b]  a = rows b = columns */

   /* allocate memory for realpatients */
   /* if not enough memory to allocate for realpatients[totalpatients] */
   if ((realpatients = (int *) malloc(totalpatients * sizeof(int))) == NULL)
{
      if (whoami == 0) {
         printf("Not enough memory to allocate for
realpatients[totalpatients].\n");
      } /* if */
#ifdef mpi
      MPI_Finalize();
#endif
      exit(1);  /* terminate program */
   } /* if */
   else {
      /* memory allocated for realpatients[totalpatients] */
      for (p = 0;  p < totalpatients; p++) {
         /* initialize all of the values in realpatients */
         realpatients[p] = 0;
      } /* for */
   } /* else */

   /* allocate memory for Gene_Accession_Number */
   /* if not enough memory to allocate for Gene_Accession_Number[totalgenes]
*/
   if ((Gene_Accession_Number = (char **) malloc(totalgenes * sizeof(char
*))) == NULL) {
      if (whoami == 0) {
         printf("Not enough memory to allocate for
Gene_Accession_Number[totalgenes].\n");
      } /* if */
#ifdef mpi
      MPI_Finalize();
#endif
      exit(1);  /* terminate program */
   } /* if */
   else {
      /* memory allocated for Gene_Accession_Number[totalgenes] */
      for (g = 0; g < totalgenes; g++) {
         /* if not enough memory to allocate for
Gene_Accession_Number[letters] */
         if ((Gene_Accession_Number[g] = (char *) malloc(letters *
sizeof(char))) == NULL) {
            if (whoami == 0) {
               printf("Not enough memory to allocate for
Gene_Accession_Number[letters].\n");
            } /* if */
#ifdef mpi
```

```
            MPI_Finalize();
#endif
            exit(1);  /* terminate program */
         } /* if */
      } /* for */
   } /* else */

   /* allocate memory for array */
   /* if not enough memory to allocate for array[genes] */
   if ((array = (double **) malloc(genes * sizeof(double *))) == NULL) {
      if (whoami == 0) {
        printf("Not enough memory to allocate for array[genes].\n");
      } /* if */
#ifdef mpi
      MPI_Finalize();
#endif
      exit(1);  /* terminate program */
   } /* if */
   else {
      /* memory allocated for array[genes] */
      for (g = 0;  g < genes; g++) {
         /* if not enough memory to allocate for array[patients] */
         if ((array[g] = (double *) malloc(patients * sizeof(double))) ==
NULL) {
            if (whoami == 0) {
               printf("Not enough memory to allocate for
array[patients].\n");
            } /* if */
#ifdef mpi
            MPI_Finalize();
#endif
            exit(1);  /* terminate program */
         } /* if */
         else {
            /* memory allocated for array[patients] */
            for (p = 0;  p < patients; p++) {
               /* initialize all of the values in array */
               array[g][p] = 0;
            } /* for */
         } /* else */
      } /* for */
   } /* else */

   /* allocate memory for call */
   /* if not enough memory to allocate for call[genes] */
   if ((call = (char **) malloc(genes * sizeof(char *))) == NULL) {
      if (whoami == 0) {
        printf("Not enough memory to allocate for call[genes].\n");
      } /* if */
#ifdef mpi
      MPI_Finalize();
#endif
      exit(1);  /* terminate program */
   } /* if */
   else {
      /* memory allocated for call[genes] */
      for (g = 0; g < genes; g++) {
         /* if not enough memory to allocate for call[patients] */
         if ((call[g] = (char *) malloc(patients * sizeof(char))) == NULL) {
            if (whoami == 0) {
              printf("Not enough memory to allocate for call[patients].\n");
            }
#ifdef mpi
            MPI_Finalize();
#endif
            exit(1);  /* terminate program */
         } /* if */
      } /* for */
   } /* else */

   /* allocate memory for z */
   /* if not enough memory to allocate for z[kmax] */
   if ((z = (double **) malloc(kmax * sizeof(double *))) == NULL) {
      if (whoami == 0) {
         printf("Not enough memory to allocate for z[kmax].\n");
      } /* if */
#ifdef mpi
      MPI_Finalize();
#endif
      exit(1);  /* terminate program */
   } /* if */
   else {
      /* memory allocated for z[kmax] */
      for (k = 0; k < kmax; k++) {
         /* if not enough memory to allocate for z[patients] */
         if ((z[k] = (double *) malloc(patients * sizeof(double))) == NULL) {
            if (whoami == 0) {
               printf("Not enough memory to allocate for z[patients].\n");
            } /* if */
#ifdef mpi
            MPI_Finalize();
#endif
            exit(1);  /* terminate program */
         } /* if */
         else {
            /* memory allocated for z[patients] */
            for (p = 0;  p < patients; p++) {
               /* initialize all of the values in z */
               z[k][p] = 0;
            } /* for */
         } /* else */
      } /* for */
   } /* else */

   /* allocate memory for netk */
   /* if not enough memory to allocate for netk[kmax] */
   if ((netk = (double **) malloc(kmax * sizeof(double *))) == NULL) {
      if (whoami == 0) {
         printf("Not enough memory to allocate for netk[kmax].\n");
      } /* if */
#ifdef mpi
      MPI_Finalize();
#endif
      exit(1);  /* terminate program */
   } /* if */
   else {
      /* memory allocated for netk[kmax] */
      for (k = 0;  k < kmax; k++) {
         /* if not enough memory to allocate for netk[patients] */
         if ((netk[k] = (double *) malloc(patients * sizeof(double))) ==
NULL) {
            if (whoami == 0) {
               printf("Not enough memory to allocate for netk[patients].\n");
            } /* if */
#ifdef mpi
            MPI_Finalize();
#endif
            exit(1);  /* terminate program */
         } /* if */
         else {
            /* memory allocated for netk[patients] */
            for (p = 0;  p < patients; p++) {
               /* initialize all of the values in netk  */
               netk[k][p] = 0;
            } /* for */
         } /* else */
      } /* for */
   } /* else */

   /* allocate memory for fnetk */
   /* if not enough memory to allocate for fnetk[kmax] */
   if ((fnetk = (double **) malloc(kmax * sizeof(double *))) == NULL) {
      if (whoami == 0) {
         printf("Not enough memory to allocate for fnetk[kmax].\n");
      } /* if */
#ifdef mpi
      MPI_Finalize();
#endif
      exit(1);  /* terminate program */
   } /* if */
   else {
      /* memory allocated for fnetk[kmax] */
      for (k = 0;  k < kmax; k++) {
         /* if not enough memory to allocate for fnetk[patients] */
         if ((fnetk[k] = (double *) malloc(patients * sizeof(double))) ==
NULL) {
            if (whoami == 0) {
               printf("Not enough memory to allocate for
fnetk[patients].\n");
            } /* if */
#ifdef mpi
            MPI_Finalize();
#endif
            exit(1);  /* terminate program */
         } /* if */
         else {
            /* memory allocated for fnetk[patients] */
            for (p = 0;  p < patients; p++) {
               /* initialize all of the values in fnetk  */
               fnetk[k][p] = 0;
            } /* for */
         } /* else */
      } /* for */
   } /* else */

   /* allocate memory for fPRIMEnetk */
   /* if not enough memory to allocate for fPRIMEnetk[kmax] */
   if ((fPRIMEnetk = (double **) malloc(kmax * sizeof(double *))) == NULL) {
      if (whoami == 0) {
         printf("Not enough memory to allocate for fPRIMEnetk[kmax].\n");
      } /* if */
#ifdef mpi
      MPI_Finalize();
#endif
      exit(1);  /* terminate program */
   } /* if */
   else {
      /* memory allocated for fPRIMEnetk[kmax] */
      for (k = 0;  k < kmax; k++) {
         /* if not enough memory to allocate for fPRIMEnetk[patients] */
         if ((fPRIMEnetk[k] = (double *) malloc(patients * sizeof(double)))
== NULL) {
            if (whoami == 0) {
               printf("Not enough memory to allocate for
fPRIMEnetk[patients].\n");
            } /* if */
#ifdef mpi
            MPI_Finalize();
#endif
            exit(1);  /* terminate program */
         } /* if */
         else {
            /* memory allocatedfor fPRIMEnetk[patients] */
            for (p = 0;  p < patients; p++) {
               /* initialize all of the values in fPRIMEnetk */
               fPRIMEnetk[k][p] = 0;
            } /* for */
         } /* else */
      } /* for */
   } /* else */

   /* allocate memory for t */
   /* if not enough memory to allocate for t[kmax] */
   if ((t = (double **) malloc(kmax * sizeof(double *))) == NULL) {
      if (whoami == 0) {
         printf("Not enough memory to allocate for t[kmax].\n");
      } /* if */
#ifdef mpi
      MPI_Finalize();
#endif
      exit(1);  /* terminate program */
   } /* if */
   else {
      /* memory allocated for t[kmax] */
      for (k = 0; k < kmax; k++) {
         /* if not enough memory to allocate for t[totalpatients] */
         if ((t[k] = (double *) malloc(totalpatients * sizeof(double))) ==
NULL) {
```

```
            if (whoami == 0) {
               printf("Not enough memory to allocate for
t[totalpatients].\n");
            } /* if */
#ifdef mpi
            MPI_Finalize();
#endif
            exit(1);  /* terminate program */
         } /* if */
         else {
            /* allocate memory for t[totalpatients] */
            for (p = 0;  p < totalpatients; p++) {
               /* initialize all of the values in t */
               t[k][p] = 0;
            } /* for */
         } /* else */
      } /* for */
   } /* else */

   /* allocate memory for DELTAk */
   /* if not enough memory to allocate for DELTAk[kmax] */
   if ((DELTAk = (double **) malloc(kmax * sizeof(double *))) == NULL) {
      if (whoami == 0) {
         printf("Not enough memory to allocate for DELTAk[kmax].\n");
      } /* if */
#ifdef mpi
      MPI_Finalize();
#endif
      exit(1);  /* terminate program */
   } /* if */
   else {
      /* memory allocated for DELTAk[kmax] */
      for (k = 0;  k < kmax; k++) {
         /* if not enough memory to allocate for DELTAk[patients] */
         if ((DELTAk[k] = (double *) malloc(patients * sizeof(double))) ==
NULL) {
            if (whoami == 0) {
               printf("Not enough memory to allocate for
DELTAk[patients].\n");
            } /* if */
#ifdef mpi
            MPI_Finalize();
#endif
            exit(1);  /* terminate program */
         } /* if */
         else {
            /* memory allocated for DELTAk[patients] */
            for (p = 0;  p < patients; p++) {
               /* initialize all of the values in DELTAk */
               DELTAk[k][p] = 0;
            } /* for */
         } /* else */
      } /* for */
   } /* else */

   /* allocate memory for x */
   /* if not enough memory to allocate for x[d] */
   if ((x = (double **) malloc(d * sizeof(double *))) == NULL) {
      if (whoami == 0) {
         printf("Not enough memory to allocate for x[d].\n");
      } /* if */
#ifdef mpi
      MPI_Finalize();
#endif
      exit(1);  /* terminate program */
   } /* if */
   else {
      /* memory allocated for x[d] */
      for (g = 0; g < d; g++) {
         /* if not enough memory to allocate for x[patients] */
         if ((x[g] = (double *) malloc(patients * sizeof(double))) == NULL) {
            if (whoami ==0) {
               printf("Not enough memory to allocate for x[patients].\n");
            } /* if */
#ifdef mpi
            MPI_Finalize();
#endif
            exit(1);  /* terminate program */
         } /* if */
         else {
            /* memory allocated for x[patients] */
            for (p = 0;  p < patients; p++) {
               /* initialize all of the values in x */
               x[g][p] = 0;
            } /* for */
         } /* else */
      } /* for */
   } /* else */

   /* allocate memory for y */
   /* if not enough memory to allocate for y[jmax] */
   if ((y = (double **) malloc(jmax * sizeof(double *))) == NULL) {
      if (whoami == 0) {
         printf("Not enough memory to allocate for y[jmax].\n");
      } /* if */
#ifdef mpi
      MPI_Finalize();
#endif
      exit(1);  /* terminate program */
   } /* if */
   else {
      /* memory allocated for y[jmax] */
      for (j = 0;  j < jmax; j++) {
         /* if not enough memory to allocate for y[patients] */
         if ((y[j] = (double *) malloc(patients * sizeof(double))) == NULL) {
            if (whoami == 0) {
               printf("Not enough memory to allocate for y[patients].\n");
            } /* if */
#ifdef mpi
            MPI_Finalize();
#endif
            exit(1);  /* terminate program */
         } /* if */
         else {
            /* memory allocated for y[patients] */
            for (p = 0;  p < patients; p++) {
               /* initialize all of the values in y */
               y[j][p] = 0;
            } /* for */
         } /* else */
      } /* for */
   } /* else */

   /* allocate memory for netj */
   /* if not enough memory to allocate for netj[jmax] */
   if ((netj = (double **) malloc(jmax * sizeof(double *))) == NULL) {
      if (whoami == 0) {
         printf("Not enough memory to allocate for netj[jmax].\n");
      } /* if */
#ifdef mpi
      MPI_Finalize();
#endif
      exit(1);  /* terminate program */
   } /* if */
   else {
      /* memory allocated for netj[jmax] */
      for (j = 0;  j < jmax; j++) {
         /* if not enough memory to allocate for netj[patients] */
         if ((netj[j] = (double *) malloc(patients * sizeof(double))) ==
NULL) {
            if (whoami == 0) {
               printf("Not enough memory to allocate for netj[patients].\n");
            } /* if */
            exit(1);  /* terminate program */
         } /* if */
         else {
            /* memory allocated for netj[patients] */
            for (p = 0;  p < patients; p++) {
               /* initialize all of the values in netj */
               netj[j][p] = 0;
            } /* for */
         } /* else */
      } /* for */
   } /* else */

   /* allocate memory for fnetj */
   /* if not enough memory to allocate for fnetj[jmax] */
   if ((fnetj = (double **) malloc(jmax * sizeof(double *))) == NULL) {
      if (whoami == 0) {
         printf("Not enough memory to allocate for fnetj[jmax].\n");
      } /* if */
#ifdef mpi
      MPI_Finalize();
#endif
      exit(1);  /* terminate program */
   } /* if */
   else {
      /* memory allocated for fnetj[jmax] */
      for (j = 0;  j < jmax; j++) {
         /* if not enough memory to allocate for fnetj[patients] */
         if ((fnetj[j] = (double *) malloc(patients * sizeof(double))) ==
NULL) {
            if (whoami == 0) {
               printf("Not enough memory to allocate for
fnetj[patients].\n");
            } /* if */
#ifdef mpi
            MPI_Finalize();
#endif
            exit(1);  /* terminate program */
         } /* if */
         else {
            /* memory allocated for fnetj[patients] */
            for (p = 0;  p < patients; p++) {
               /* initialize all of the values in fnetj */
               fnetj[j][p] = 0;
            } /* for */
         } /* else */
      } /* for */
   } /* else */

   /* allocate memory for fPRIMEnetj */
   /* if not enough memory to allocate for fPRIMEnetj[jmax] */
   if ((fPRIMEnetj = (double **) malloc(jmax * sizeof(double *))) == NULL) {
      if (whoami == 0) {
         printf("Not enough memory to allocate for fPRIMEnetj[jmax].\n");
      } /* if */
#ifdef mpi
      MPI_Finalize();
#endif
      exit(1);  /* terminate program */
   } /* if */
   else {
      /* memory allocated for fPRIMEnetj[jmax] */
      for (j = 0;  j < jmax; j++) {
         /* if not enough memory to allocate for fPRIMEnetj[patients] */
         if ((fPRIMEnetj[j] = (double *) malloc(patients * sizeof(double)))
== NULL) {
            if (whoami == 0) {
               printf("Not enough memory to allocate for
fPRIMEnetj[patients].\n");
            } /* if */
#ifdef mpi
            MPI_Finalize();
#endif
            exit(1);  /* terminate program */
         } /* if */
         else {
            /* memory allocated for fPRIMEnetj[patients] */
            for (p = 0;  p < patients; p++) {
               /* initialize all of the values in fPRIMEnetj */
               fPRIMEnetj[j][p] = 0;
            } /* for */
         } /* else */
      } /* for */
   } /* else */
   /* allocate memory for wji */
   /* if not enough memory to allocate for wji[jmax] */
   if ((wji = (double ***) malloc(jmax * sizeof(double **))) == NULL) {
      if (whoami == 0) {
         printf("Not enough memory to allocate for wji[jmax].\n");
      } /* if */
#ifdef mpi
      MPI_Finalize();
#endif
      exit(1);  /* terminate program */
   } /* if */
   else {
      /* memory allocated for wji[jmax] */
      for (j = 0;  j < jmax; j++) {
         /* if not enough memory to allocate for wji[d] */
         if ((wji[j] = (double **) malloc(d * sizeof(double *))) == NULL) {
            if (whoami == 0) {
               printf("Not enough memory to allocate for wji[d].\n");
            } /* if */
#ifdef mpi
            MPI_Finalize();
#endif
            exit(1);  /* terminate program */
         } /* if */
         else {
            /* memory allocated for wji[d] */
            for (i = 0;  i < d; i++) {
               /* if not enough memory to allocate for wji[patients] */
               if ((wji[j][i] = (double *) malloc(patients * sizeof(double)))
== NULL) {
                  if (whoami == 0) {
                     printf("Not enough memory to allocate for
wji[patients].\n");
                  }
#ifdef mpi
                  MPI_Finalize();
#endif
                  exit(1);  /* terminate program */
               } /* if */
               else {
                  /* memory allocated for wji[patients] */
                  for (p = 0;  p < patients; p++) {
                     /* initialize all of the values in wji */
                     wji[j][i][p] = 1;
/* printf("wji[%d][%4d][%2d] = %8lf\n", j, i, p, wji[j][i][p]); */
                  } /* for */
               } /* else */
            } /* for */
         } /* else */
      } /* for */
   } /* else */

   /* allocate memory for DELTAwji */
   /* if not enough memory to allocate for DELTAwji[jmax] */
   if ((DELTAwji = (double **) malloc(jmax * sizeof(double *))) == NULL) {
      if (whoami == 0) {
         printf("Not enough memory to allocate for DELTAwji[jmax].\n");
      } /* if */
#ifdef mpi
      MPI_Finalize();
#endif
      exit(1);  /* terminate program */
   } /* if */
   else {
      /* memory allocated for DELTAwji[jmax] */
      for (j = 0;  j < jmax; j++) {
         /* if not enough memory to allocate for DELTAwji[d] */
         if ((DELTAwji[j] = (double *) malloc(d * sizeof(double))) == NULL) {
            if (whoami == 0) {
               printf("Not enough memory to allocate for DELTAwji[d].\n");
            } /* if */
#ifdef mpi
            MPI_Finalize();
#endif
            exit(1);  /* terminate program */
         } /* if */
         else {
            /* memory allocated for DELTAwji[d] */
            for (i = 0;  i < d; i++) {
               /* initialize all of the values in DELTAwji */
               DELTAwji[j][i] = 0;
            } /* for */
         } /* else */
      } /* for */
   } /* else */

   /* allocate memory for wkj */
   /* if not enough memory to allocate for wkj[kmax] */
   if ((wkj = (double ***) malloc(kmax * sizeof(double **))) == NULL) {
      if (whoami == 0) {
        printf("Not enough memory to allocate for wkj[kmax].\n");
      } /* if */
#ifdef mpi
      MPI_Finalize();
#endif
      exit(1);  /* terminate program */
   } /* if */
   else {
      /* memory allocated for wkj[kmax] */
      for (k = 0;  k < kmax; k++) {
         /* if not enough memory to allocate for wkj[jmax] */
         if ((wkj[k] = (double **) malloc(jmax * sizeof(double *))) == NULL)
{
            if (whoami == 0) {
              printf("Not enough memory to allocate for wkj[jmax].\n");
            } /* if */
#ifdef mpi
            MPI_Finalize();
#endif
            exit(1);  /* terminate program */
         } /* if */
         else {
            /* memory allocated for wkj[jmax] */
            for (j = 0;  j < jmax; j++) {
               /* if not enough memory to allocate for wkj[patients] */
               if ((wkj[k][j] = (double *) malloc(patients * sizeof(double)))
== NULL) {
                  if (whoami == 0) {
                    printf("Not enough memory to allocate for
wkj[patients].\n");
                  } /* if */
#ifdef mpi
                  MPI_Finalize();
#endif

                  exit(1);  /* terminate program */
               } /* if */
               else {
                  /* memory allocated for wkj[patients] */
                  for (p = 0;  p < patients; p++) {
                     /* initialize all of the values in wkj */
                     wkj[k][j][p] = 2;
                  } /* for */
               } /* else */
            } /* for */
         } /* else */
      } /* for */
   } /* else */

   /* allocate memory for DELTAwkj */
   /* if not enough memory to allocate for DELTAwkj[kmax] */
   if ((DELTAwkj = (double **) malloc(kmax * sizeof(double *))) == NULL) {
      if (whoami == 0) {
        printf("Not enough memory to allocate for DELTAwkj[kmax].\n");
      } /* if */
#ifdef mpi
      MPI_Finalize();
#endif
      exit(1);  /* terminate program */
   } /* if */
   else {
      /* memory allocated for DELTAwkj[kmax] */
      for (k = 0;  k < kmax; k++) {
         /* if not enough memory to allocate for DELTAwkj[jmax] */
         if ((DELTAwkj[k] = (double *) malloc(jmax * sizeof(double))) ==
NULL) {
            if (whoami == 0) {
              printf("Not enough memory to allocate for DELTAwkj[jmax].\n");
            } /* if */
#ifdef mpi
            MPI_Finalize();
#endif
            exit(1);  /* terminate program */
         } /* if */
         else {
            /* memory allocated for DELTAwkj[jmax] */
            for (j = 0;  j < jmax; j++) {
               /* initialize all of the values in DELTAwkj */
               DELTAwkj[k][j] = 0;
            } /* for */
         } /* else */
      } /* for */
   } /* else */
} /* allocate_dynamic_array_memory */

void input_layer(FILE* fp) {
   if (preprocessing == 0 || preprocessing == 4) {
      if (!strcmp (trainortest, "train")) {
         for (p = 0; p < npat; p++) {
            for (g = 0; g < genes; g++) {
               for (cn = 0; cn < 50; cn++) {
                  if (!strcmp (fiftygenes[cn], Gene_Accession_Number[g])) {
                     q = realpatients[p];
                     /* assign values to x */
                     x[cn][q] = array[g][q];
                  } /* if */
               } /* for */
            } /* for */
         } /* for */
      } /* if */
      else { /* if (!strcmp (trainortest, "test")) { */
         for (g = genes; g < totalgenes; g++) {
            for (cn = 0; cn < 50; cn++) {
               if (!strcmp (fiftygenes[cn], Gene_Accession_Number[g])) {
                  for (p = 0; p < npat; p++) {
                     q = realpatients[p] - inputpatients;
                     j = g - genes;
                     /* assign values to x */
                     x[cn][q] = array[j][q];
                  } /* for */
               } /* if */
            } /* for */
         } /* for */
      } /* else */
   } /* if */

   else { /* if (preprocessing == 1 || preprocessing == 2 || preprocessing ==
3 || preprocessing == 5) */
      if (!strcmp (trainortest, "train")) {
         for (p = 0; p < npat; p++) {
            for (g = 0; g < genes; g++) {
               q = realpatients[p];
               x[g][q] = array[g][q];
            } /* for */
         } /* for */
      } /* if */
      else { /* if (!strcmp (trainortest, "test")) { */
         for (p = 0; p < npat; p++) {
            for (g = genes; g < totalgenes; g++) {
               q = realpatients[p] - inputpatients;
               x[g][q] = array[g][q];
            } /* for */
         } /* for */
      } /* else */
   } /* else */

/* If this is the first time through, compute/store baseline information,
which is    from the first patient in the training dataset. */
```

```
   if(!strcmp (trainortest, "train")) {
      compute_baseline();
   } /* if */

   rescale(); /* Use computed/stored baseline data to rescale patient data */
} /* input_layer */

void compute_baseline() {
   mu1 = 0.0;
   for (g = 0; g < d; g++) {
      mu1 += x[g][0];
   } /* for */
   mu1 = mu1 / d;
} /* compute_baseline */

void rescale() {
/* re-scales patient columns of gene expression data with respect to patient
#1 of training set; then normalizes each patient column using the mean and
standard deviation computed for that column.   This procedure is based on a
reconstruction of the MIT group's normalization procedure as described in
their Supplementary Information, www.genome.wi.mit.edu/MPR, and normalization
and scaling procedures from the Affymetrix GeneChip Expression Analysis
Technical Manual, ch. 5.  */

   double sigma;        /* standard deviation of gene expression values for a
given patient */
   double mu;           /* mean gene expression value for a given patient  */
   double temp;         /* temporary variable */

   /* compute mean mu and standard deviation sigma for each patient p;
      re-scale expression values via x[g][p] = x[g][p] * mu1/mu;
      "z-score" re-scaled values via x[g][p] = (x[g][p] - mu1)/sigma  */

   for (q = 0; q < npat; q++) {
      mu = 0.0;
      for (g = 0; g < d; g++) {
         mu += x[g][q];
      } /* for */
      mu = mu/d;

      sigma = 0.0;
      for (g = 0; g < d; g++) {
         temp   = x[g][q] - mu;
         sigma += temp*temp;
      } /* for */
      sigma = sqrt (sigma / (d - 1));

      for (g = 0; g < d; g++) {
         x[g][q] = x[g][q] * mu1 / mu;
         x[g][q] = (x[g][q] - mu1) / sigma;
      } /* for */
   } /* for */
} /* rescale */

/* deletes all values for gene expression identical for every patient */
void preprocessing1() {
   for (g = 0; g < d; g++) {
      blankint = 0;
      if (!strcmp (trainortest, "train")) {
         for (p = 0; p < inputpatients; p++) {
            if (p == 0) {
               x[g][p] = blankint;
            } /* if */
            else if (x[g][p] != blankint) {
               g++;
               goto evaluation1; /* 2 level continue – outer for */
            } /* else if */
         } /* for */
         for (p = 0; p < inputpatients; p++) {
            x[g][p] = 0;
         } /* for */
         evaluation1: ; /* null statement; continue loop */
      } /* if */
      else if (!strcmp (trainortest, "test")) {
         for (p = 0; p < testingpatients; p++) {
            if (p == 0) {
               x[g][p] = blankint;
            } /* if */
            else if (x[g][p] != blankint) {
               g++;
               goto evaluation2; /* 2 level continue – outer for */
            } /* else if */
         } /* for */
         for (p = 0; p < testingpatients; p++) {
            x[g][p] = 0;
         } /* for */
         evaluation2: ; /* null statement; continue loop */
      } /* else if */
   } /* for */
   if (preprocessing == 3) {
      preprocessing2();
   } /* if */
} /* preprocessing1 */

void preprocessing2() {
   for (g = 0; g < d; g++) {
      Acounter = 0;
      Pcounter = 0;
      Mcounter = 0;
      if (!strcmp (trainortest, "train")) {
         for (p = 0; p < inputpatients; p++) {
            if (!strcmp(&call[g][p], "A")) { /* A = absent */
               Acounter++;
            } /* if */
            else if(!strcmp(&call[g][p], "P")) { /* P = present */
               Pcounter++;
            } /* else if */
            else if(!strcmp(&call[g][p], "M")) { /* M = marginal */
               Mcounter++;
            } /* else if */
         } /* for */
         if (Acounter == inputpatients || Pcounter == inputpatients ||
Mcounter == inputpatients) {
            if (!strcmp (trainortest, "train")) {
               for (p = 0; p < inputpatients; p++) {
                  x[g][p] = 0;
               } /* for */
            } /* if */
         } /* if */
         if (Acounter == inputpatients || Pcounter == inputpatients ||
Mcounter == inputpatients) {
            for (p = 0; p < inputpatients; p++) {
               x[g][p] = 0;
            } /* for */
         } /* if */
      } /* if */
      else if (!strcmp (trainortest, "test")) {
         for (p = 0; p < testingpatients; p++) {
            if (!strcmp(&call[g][p], "A")) { /* A = absent */
               Acounter++;
            } /* if */
            else if(!strcmp(&call[g][p], "P")) { /* P = present */
               Pcounter++;
            } /* else if */
            else if(!strcmp(&call[g][p], "M")) { /* M = marginal */
               Mcounter++;
            } /* else if */
         } /* for */
         if (Acounter == testingpatients || Pcounter == testingpatients ||
Mcounter == testingpatients) {
            if (!strcmp (trainortest, "train")) {
               for (p = 0; p < inputpatients; p++) {
                  x[g][p] = 0;
               } /* for */
            } /* if */
         } /* if */
         if (Acounter == testingpatients || Pcounter == testingpatients ||
Mcounter == testingpatients) {
            for (p = 0; p < testingpatients; p++) {
               x[g][p] = 0;
            } /* for */
         } /* if */
      } /* else if */
   } /* for */
} /* preprocessing2 */

void preprocessing3() {
   /* allocate memory for fiftygenes */
   for (g = 0; g < 50; g++) {
      /* if not enough memory to allocate for fiftygenes[letters] */
      if ((fiftygenes[g] = (char *) malloc(letters * sizeof(char))) == NULL)
{
         printf("Not enough memory to allocate for fiftygenes[letters].\n");
         terminate();  /* terminate program */
      } /* if */
      else {
         /* memory allocated for fiftygenes[letters] */
         /* assign values to fiftygenes */
         fiftygenes[0] = "U22376_cds2_s_at";
         fiftygenes[1] = "X59417_at";
         fiftygenes[2] = "U05259_rna1_at";
         fiftygenes[3] = "M92287_at";
         fiftygenes[4] = "M31211_s_at";
         fiftygenes[5] = "X74262_at";
         fiftygenes[6] = "D26156_s_at";
         fiftygenes[7] = "S50223_at";
         fiftygenes[8] = "M31523_at";
         fiftygenes[9] = "L47738_at";
         fiftygenes[10] = "U32944_at";
         fiftygenes[11] = "Z15115_at";
         fiftygenes[12] = "X15949_at";
         fiftygenes[13] = "X63469_at";
         fiftygenes[14] = "M91432_at";
         fiftygenes[15] = "U29175_at";
         fiftygenes[16] = "Z69881_at";
         fiftygenes[17] = "U20998_at";
         fiftygenes[18] = "D38073_at";
         fiftygenes[19] = "U26266_s_at";
         fiftygenes[20] = "M31303_rna1_at";
         fiftygenes[21] = "Y08612_at";
         fiftygenes[22] = "U35451_at";
         fiftygenes[23] = "M29696_at";
         fiftygenes[24] = "M13792_at";
         fiftygenes[25] = "M55150_at";
         fiftygenes[26] = "X95735_at";
         fiftygenes[27] = "U50136_rna1_at";
         fiftygenes[28] = "M16038_at";
         fiftygenes[29] = "U82759_at";
         fiftygenes[30] = "M23197_at";
         fiftygenes[31] = "M84526_at";
         fiftygenes[32] = "Y12670_at";
         fiftygenes[33] = "M27891_at";
         fiftygenes[34] = "X17042_at";
         fiftygenes[35] = "Y00787_s_at";
         fiftygenes[36] = "M96326_rna1_at";
         fiftygenes[37] = "U46751_at";
         fiftygenes[38] = "M80254_at";
         fiftygenes[39] = "L08246_at";
         fiftygenes[40] = "M62762_at";
         fiftygenes[41] = "M28130_rna1_s_at";
         fiftygenes[42] = "M63138_at";
         fiftygenes[43] = "M57710_at";
         fiftygenes[44] = "M69043_at";
         fiftygenes[45] = "M81695_s_at";
         fiftygenes[46] = "X85116_rna1_s_at";
         fiftygenes[47] = "M19045_f_at";
         fiftygenes[48] = "M83652_s_at";
         fiftygenes[49] = "X04085_rna1_at";
      } /* else */
   } /* for */
} /* preprocessing3 */

void hidden_layer() {
   /* calculate netj */
   /* netj = d_SUMMATION_i=0 xi * wji  (1) */
   for (j = 0; j < jmax; j++) {
      for (i = 0; i < d; i++) {
         netj[j][p] = netj[j][p] + x[i][p] * wji[j][i][p];
      } /* for */
   } /* for */

   /* calculate f(netj) */
   /* f(netj) = 2a / (1 + e^-bnetj) - a  (34) */
   for (j = 0; j < jmax; j++) {
      if (j == 0 && p == 0) {
         fnetj[j][p] = 1.0;
      } /* if */
      else {
         fnetj[j][p] = ((2 * a) / (1 + exp(-b * netj[j][p])) - a); */
      } /* else */
   } /* for */

   /* calculate yj */
   /* yj = f(netj)  (2) */
   for (j = 0; j < jmax; j++) {
      y[j][p] = fnetj[j][p];
   } /* for */
} /* hidden_layer */

void output_layer() {
   /* calculate netk */
   /* netk = n[H]_SUMMATION_j=0 yj wkj  (4) */
   /* jmax = nH; */
   for (k = 0; k < kmax; k++) {
      netk[k][p] = 0;
      for (j = 0; j < jmax; j++) { /* nH */
         netk[k][p] = netk[k][p] + y[j][p] * wkj[k][j][p];
      } /* for */
   } /* for */

   /* calculate f(netk) */
   /* f(netk) = 2a / (1 + e^(-bnetk)) - a  (34) */
   for (k = 0; k < kmax; k++) {
      fnetk[k][p] = ((2 * a) / (1 + exp(-b * netk[k][p])) - a);
   } /* for */

   /* calculate zk */
   /* zk = f(netk)  (5) */
   for (k = 0; k < kmax; k++) {
      z[k][p] = fnetk[k][p];
   } /* for */
} /* output_layer */

void backpropagation() {
   /* calculate f'(netk) */
   /* f'(netk) = [2abe^(-b*netk)]/[1+2e^(-b*netk)+e^(-2b*netk)] */
   for (k = 0; k < kmax; k++) {
      fPRIMEnetk[k][p] = (2 * a * b * exp(-b * netk[k][p])) / (1 + 2 * exp(-b
* netk[k][p]) + exp(2 * -b * netk[k][p]));
   } /* for */

   /* calculate DELTAwkj */
   /* DELTAwkj = ETA * (tk - zk) * f'(netk) * yj  (17)  */
   for (j = 0; j < jmax; j++) {
      for (k = 0; k < kmax; k++) { /* c */
```

```
         DELTAwkj[k][j] = DELTAwkj[k][j] + ETA * (t[k][p] - z[k][p]) *
fPRIMEnetk[k][p] * y[j][p];
      } /* for */
   } /* for */

   /* calculate f'(netj) */
   /* f'(netj) = (2abe^(-bx)) / (1 + 2e^(-bx) + e^(-2bx)) */
   for (j = 0; j < jmax; j++) {
      fPRIMEnetj[j][p] = (2 * a * b * exp(-b * netj[j][p])) / (1 + 2 * exp(-b
* netj[j][p]) + exp(2 * -b * netj[j][p]));
   } /* for */

   /* calculate DELTAk */
   /* DELTAk = (tk - zk) * f'(netk)  (15) */
   for (k = 0; k < kmax; k++) { /* c */
      DELTAk[k][p] = (t[k][p] - z[k][p]) * fPRIMEnetk[k][p];
   } /* for */

   /* calculate DELTAwji */
   /* DELTAwji = ETA[c_SUMMATION_k=1 wkj * DELTAk * f'(netj) * xi  (21) */
   for (i = 0; i < d; i++) {
      for (j = 0; j < jmax; j++) {
         for (k = 0; k < kmax; k++) { /* c */
            DELTAwji[j][i] = DELTAwji[j][i] + ETA * (DELTAwji[j][i] +
wkj[k][j][p] * DELTAk[k][p]) * fPRIMEnetj[j][p] * x[i][p];
         } /* for */
      } /* for */
   } /* for */
} /* backpropagation */

void update_weights() {
   /* calculate wkj(m+1) */
   /* wkj = wkj + DELTAwkj */
   /* wkj(m+1) = wkj(m) + DELTAwkj(m)  (12)  */
   for (k = 0; k < kmax; k++) { /* m + 1 */
      for (j = 0; j < jmax; j++) { /* m + 1 */
         wkj[k][j][p] = wkj[k][j][p] + DELTAwkj[k][j];
      } /* for */
   } /* for */

   /* calculate wji(m+1) */
   /* wji(m+1) = wji(m) + DELTAwji(m)  (12) */
   for (j = 0; j < jmax; j++) { /* m + 1 */
      for (i = 0; i < d; i++) { /* m + 1 */
         wji[j][i][p] = wji[j][i][p] + DELTAwji[j][i];
      } /* for */
   } /* for */
} /* update_weights */

/* specialized subroutine for classification of two subtypes; only 1 output
node */
void print_subtypes(FILE* fp) {
   fprintf(fp, "Patient   Subtype     Number    Type of    Learning
Number of\n");
   fprintf(fp, "Number  Designation  of Genes  Training      Rate       Nodes
in\n");
```

```
   fprintf(fp, "                      Used As    Used                 the
Hidden\n");
   fprintf(fp, "                       Input
Layer\n");
   for (p = 0; p < testingpatients; p++) {
      for (k = 0; k < kmax; k++) {
         if (p == 0) {
            if (z[k][p] == -1) { /* -1 = ALL */
               fprintf(fp, "  %2d\t   AML\t      %d\t     %s\t  %lf\t
%d\n", p, genes, trainingtype, ETA, jmax); /* print patients ALL */
            } /* if */
            else if (z[k][p] == 1) { /* +1 = AML */
               fprintf(fp, "  %2d\t   ALL\t      %d\t     %s\t  %lf\t
%d\n", p, genes, trainingtype, ETA , jmax); /* printpatients AML */
            } /* else if */
            else {
               fprintf(fp, "  %2d\t   %8lf\t      %d\t     %s\t  %lf\t
%d\n", p, z[k][p], genes, trainingtype, ETA , jmax);
            } /* else */
         } /* if */
         if (z[k][p] == -1) { /* -1 = ALL */
            fprintf(fp, "  %2d\t   AML\n", p); /* print patients ALL */
         } /* if */
         else if (z[k][p] == 1) { /* +1 = AML */
            fprintf(fp, "  %2d\t   ALL\n", p); /* print patients AML */
         } /* else if */
         else {
            fprintf(fp, "  %2d\t   %8lf\n", p, z[k][p]);
         } /* else */
      } /* for */
   } /* for */
} /* print_subtypes */

void deallocate_dynamic_array_memory() {
   /* deallocate memory for array */
   for (g = 0; g < genes; g++) {
      /* deallocate memory for array[patients] */
      free(array[g]);
   } /* for */

   /* deallocate memory for array[genes] */
   free(array);

   /* deallocate memory for y */
   for (j = 0; j < jmax; j++) {
      free(y[j]);
   }
   /* deallocate memory for y[jmax] */
   free(y);

   /* deallocate memory for wji */
   for (j = 0; j < jmax; j++) {
      for (i = 0;  i < d; i++) {
         /* deallocate memory for wji[patients] */
         free(wji[j][i]);
      } /* for */
   } /* for */
```

```
for (j = 0; j < jmax; j++) {
   /* deallocate memory for wji[d] */
   free(wji[j]);
} /* for */
/* deallocate memory for wji[jmax] */
free(wji);

/* deallocate memory for DELTAwji */
for (j = 0; j < jmax; j++) {
   /* deallocate memory for DELTAwji[d] */
   free(DELTAwji[j]);
} /* for */
/* deallocate memory for DELTAwji[jmax] */
free(DELTAwji);

/* deallocate memory for netj */
for (j = 0; j < jmax; j++) {
   /* deallocate memory for netj[patients] */
   free(netj[j]);
} /* for */
/* deallocate memory for netj[jmax] */
free(netj);

/* deallocate memory for fnetj */
for (j = 0; j < jmax; j++) {
   /* deallocate memory for fnetj[patients] */
   free(fnetj[j]);
} /* for */
/* deallocate memory for fnetj[jmax] */
free(fnetj);

/* deallocate memory for fPRIMEnetj */
for (j = 0; j < jmax; j++) {
   /* if not enough memory to allocate for fPRIMEnetj[patients] */
   free(fPRIMEnetj[j]);
} /* for */
/* deallocate memory for fPRIMEnetj[jmax] */
free(fPRIMEnetj);

/* deallocate memory for z */
for (k = 0; k < kmax; k++) {
   /* deallocate memory for z[patients] */
   free(z[k]);
} /* for */
/* deallocate memory for z[kmax] */
free(z);

/* deallocate memory for wkj */
for (k = 0;  k < kmax; k++) {
   for (j = 0;  j < jmax; j++) {
      /* deallocate memory for wkj[patients] */
      free(wkj[k][j]);
   } /* for */
} /* for */
for (k = 0;  k < kmax; k++) {
   /* deallocate memory for wkj[jmax] */
   free(wkj[k]);
} /* for */
/* deallocate memory for wkj[kmax] */
free(wkj);

/* deallocate memory for DELTAwkj */
for (k = 0;  k < kmax; k++) {
   /* deallocate memory for DELTAwkj[jmax] */
   free(DELTAwkj[k]);
} /* for */
/* deallocate memory for DELTAwkj[kmax] */
free(DELTAwkj);

/* deallocate memory for netk */
for (k = 0;  k < kmax; k++) {
   /* deallocate memory for netk[patients] */
   free(netk[k]);
} /* for */
/* deallocate memory for netk[kmax] */
free(netk);

/* deallocate memory for fPRIMEnetk */
/* deallocate memory for fPRIMEnetk[patients] */
for (k = 0;  k < kmax; k++) {
   /* deallocate memory for fPRIMEnetk[patients] */
   free(fPRIMEnetk[k]);
} /* for */
/* deallocate memory for fPRIMEnetk[kmax] */
free(fPRIMEnetk);

/* deallocate memory for t */
for (s = 0; s < kmax; s++) {
   /* deallocate memory for t[patients] */
   free(t[s]);
} /* for */
/* deallocate memory for t[kmax] */
free(t);

/* deallocate memory for realpatients */
/* deallocate memory for realpatients[patients] */
free(realpatients);

if (preprocessing == 4 || preprocessing == 0) {
   /* deallocate memory for fiftygenes */
   for (g = 49; g > 0; g--) {
      /* deallocate memory for fiftygenes[g] */
      free(fiftygenes[g]);
   } /* for */
} /* if */

/* deallocate memory for x */
for (g = 0; g < d; g++) {
/* deallocate memory for x[patients] */
  free(x[g]);
} /* for */

/* deallocate memory for x[d] */
free(x);
```

```
} /* deallocate_dynamic_array_memory */

void terminate() {
    /* clean up and exit from all processors */
    deallocate_dynamic_array_memory();
#ifdef mpi
    MPI_Finalize();
#endif
    exit(0); /* terminate program */
}; /* terminate */
```

Appendix D: Other Code Components

Makefile

```
#-------------------------------------------------------------------
#   makefile for nnclass.c
#-------------------------------------------------------------------
#   compiler, precompiler, optimization, library, and #defines
#-------------------------------------------------------------------

PGF_LIN  = mpicc

# if want PGI compiler without parallel stuff included
#PGF_LIN  = pgcc

GNU_LIN  = gcc
COMPILER = $(GNU_LIN)

# PGI compiler options from
http://www.pgroup.com/ppro_docs/pgiws_ug/pgi32u.htm,
#   Section 7, command-line options
# GNU compiler options from 'man gcc'

OPT_PGF  = -fast
OPT_GNU  = -g
OPTS     = $(OPT_GNU)

#library

LIB_MATH = -lm
LIBS     = $(LIB_MATH)

# #defines, if parallel, want to define variable MPI
DMPI     = -Dmpi
DPMODEL  =

#-------------------------------------------------------------------
#  compile, link
#-------------------------------------------------------------------

OBJS     = nnclass.o timers.o

nnclass: $(OBJS)
        $(COMPILER) $(OPTS) -o nnclass $(OBJS) $(LIBS)

nnclass.o: nnclass.c
        $(COMPILER) $(DPMODEL) -c $(OPTS) nnclass.c

timers.o: timers.c
        $(COMPILER) $(DPMODEL) -c $(OPTS) timers.c

#-------------------------------------------------------------------
#  clean-up
#-------------------------------------------------------------------

clean:
        /bin/rm *.o
```

timers.c

```
/* wrapper routines for wtime */

#include "nntimers.h"
double start[20];
double elapsed[20];
int nstarts[20];


/************************************************************************
*
*     Module name wtime.c  Version 1.0  Date 9/30/95.
*     (variation of standard wtime.c from AHPCC, sra).
*
***********************************************************************/

#include <sys/time.h>
#include <unistd.h>

void wtime(double * stime)
{
        struct timeval tb;
        struct timezone tz;
        int iret;
        iret = gettimeofday(&tb,&tz);
        if (iret!=0) printf("gettimeofday failed\n");
        *stime = (double) ((double)(tb.tv_sec) + (double)(tb.tv_usec)*1.0e-
6);
}

/*---------------------------------------------------------------------*/
      void timer_clear(int timer) {
/*---------------------------------------------------------------------*/

      start[timer]   =  0.0;
      elapsed[timer] =  0.0;
      nstarts[timer] =    0;

      }

/*---------------------------------------------------------------------*/
      void timer_start(int timer) {
/*---------------------------------------------------------------------*/

#ifdef mpi
#include "mpi.h"
      nstarts[timer] = nstarts[timer] + 1;
      start[timer]   = MPI_Wtime();
#else
      nstarts[timer] = nstarts[timer] + 1;
      wtime(&start[timer]);
#endif
      }

/*----------------------------------------------------------------------
-*/
      void timer_stop(int timer) {
```

```
/*---------------------------------------------------------------------------
-*/

#ifdef mpi
#include "mpi.h"
      elapsed[timer]+=  (MPI_Wtime() - start[timer]);

#else
      double end;
      wtime(&end);
      elapsed[timer]+=  (end - start[timer]);
#endif
      }
```

nntimers.h

```
/* global arrays for timers */

extern double start[];
extern double elapsed[];
extern int nstarts[];

void timer_clear(int timer);
void timer_start(int timer);
void timer_stop(int timer);
void wtime(double * stime);
```

nnclass.pbs

```
#!/bin/csh
#PBS -l nodes=4
#PBS -l walltime=1:00:00

setenv NODE_FILE $PBS_NODEFILE
setenv RUN_DIR /users/susie/geneteam/run
set ntasks = 4
set nodes = `cat ${NODE_FILE}`
set ID = 999

@ i = 0
foreach node ($nodes)
      @ i++
end
@ nnodes = $I

echo "MPIHOME = $MPIHOME"
echo "nnodes = $nnodes"
echo "nprocs = $ntasks"

# use gmpiconf2 for 2 processors/node; gmpiconf for 1 processor/node
# gmpiconf2 ${NODE_FILE}

if($ntasks == $nnodes) then
      echo "executing gmpiconf for 1 processor/node...."
      set smp = 1
      gmpiconf ${NODE_FILE}
else
      echo "executing gmpiconf2 for 2 processors/node...."
      set smp = 2
      gmpiconf2 ${NODE_FILE}
endif

echo " "
echo "=============== Program start ==============="
echo " "

touch nnclass$ID.log
echo " " >> nnclass$ID.log;
echo " " >> nnclass$ID.log
echo "=============== NNCLASS LOG ==============" >> nnclass$ID.log
echo " " >> nnclass$ID.log
echo "Physical nodes for this job: " >> nnclass$ID.log
echo " " >> nnclass$ID.log
cat $PBS_NODEFILE >> nnclass$ID.log

################################### run #################################

set execstart = `date`
mpirun -np $ntasks -machinefile $PBS_NODEFILE nnclass nnclassparam.dat
data_set_ALL_AML_train.txt data_set_ALL_AML_independent.txt
table_ALL_AML_samples.txt nnclass.out
set execend = `date`

################################## cleanup ################################
```

```
echo " "
echo "=============== Program end ==============="
echo " "
echo " " >> nnclass$ID.log
echo " nnclass start: $execstart" >> nnclass$ID.log echo " nnclass end:
$execend" >> nnclass$ID.log
echo " " >> nnclass$ID.log
```

Appendix E: Input Data Sets

nnclassparam.dat

```
1
0
0
batch
```

data_set_ALL_AML_train.txt

```
Gene Description  Gene Accession Number   1     call  2     call  3     call  4     call  5     call  6
      call  7     call  8     call  9     call  10    call  11    call  12    call  13    call  14    call
      15    call  16    call  17    call  18    call  19    call  20    call  21    call  22    call  23
      call  24    call  25    call  26    call  27    call  34    call  35    call  36    call  37    call
      38    call  28    call  29    call  30    call  31    call  32    call  33    call
AFFX-BioB-5_at (endogenous control) AFFX-BioB-5_at    -214  A     -139  A     -76   A     -135  A     -106
      A     -138  A     -72   A     -413  A     5     A     -88   A     -165  A     -67   A     -92   A
      -113  A     -107  A     -117  A     -476  A     -81   A     -44   A     17    A     -144  A     -247
      A     -74   A     -120  A     -81   A     -112  A     -273  A     -20   A     7     A     -213  A
      -25   A     -72   A     -4    A     15    A     -318  A     -32   A     -124  A     -135  A
AFFX-BioB-M_at (endogenous control) AFFX-BioB-M_at    -153  A     -73   A     -49   A     -114  A     -125
      A     -85   A     -144  A     -260  A     -127  A     -105  A     -155  A     -93   A     -119  A
      -147  A     -72   A     -219  A     -213  A     -150  A     -51   A     -229  A     -199  A     -90
      A     -321  A     -263  A     -150  A     -233  A     -327  A     -207  A     -100  A     -252  A
      -20   A     -139  A     -116  A     -114  A     -192  A     -49   A     -79   A     -186  A
AFFX-BioB-3_at (endogenous control) AFFX-BioB-3_at    -58   A     -1    A     -307  A     265   A     -76
      A     215   A     238   A     7     A     106   A     42    A     -71   A     84    A     -31   A
      -118  A     -126  A     -50   A     -18   A     -119  A     100   A     79    A     -157  A     -168
      A     -11   A     -114  A     -85   A     -78   A     -76   A     -50   A     -57   A     136   A
      124   A     -1    A     -125  A     2     A     -95   A     49    A     -37   A     -70   A
AFFX-BioC-5_at (endogenous control) AFFX-BioC-5_at    88    A     283   A     309   A     12    A     168
      A     71    A     55    A     -2    A     268   A     219   M     82    A     25    A     173   A
      243   M     149   A     257   A     301   A     78    A     207   A     218   A     132   A     -24
      A     -36   A     255   A     316   A     54    A     81    A     101   A     132   A     318   A
      325   A     392   P     241   A     193   A     312   A     230   P     330   A     337   A
AFFX-BioC-3_at (endogenous control) AFFX-BioC-3_at    -295  A     -264  A     -376  A     -419  A     -230
      A     -272  A     -399  A     -541  A     -210  A     -178  A     -163  A     -179  A     -233  A
      -127  A     -205  A     -218  A     -403  A     -152  A     -146  A     -262  A     -151  A     -308
      A     -317  A     -342  A     -418  A     -244  A     -439  A     -369  A     -377  A     -209  A
      -396  A     -324  A     -191  A     -51   A     -139  A     -367  A     -188  A     -407  A
AFFX-BioDn-5_at (endogenous control)      AFFX-BioDn-5_at   -558  A     -400  A     -650  A     -585  A
      -284  A     -558  A     -551  A     -790  A     -535  A     -246  A     -430  A     -323  A     -227
      A     -398  A     -284  A     -402  A     -394  A     -340  A     -221  A     -404  A     -347  A
      -571  A     -499  A     -396  A     -461  A     -275  A     -616  A     -529  A     -478  A     -557
      A     -464  A     -510  A     -411  A     -155  A     -344  A     -508  A     -423  A     -566  A
```

NOTE: This is only the 1st seven lines of this independent data set. The complete independent data set is 7130 lines in length.

data_set_ALL_AML_independent.txt

```
Gene Description  Gene Accession Number   39    call  40    call  42    call  47    call  48    call  49
      call  41    call  43    call  44    call  45    call  46    call  70    call  71    call  72    call
      68    call  69    call  67    call  55    call  56    call  59    call  52    call  53    call  51
      call  50    call  54    call  57    call  58    call  60    call  61    call  65    call  66    call
      63    call  64    call  62    call
AFFX-BioB-5_at (endogenous control) AFFX-BioB-5_at    -342  A     -87   A     22    A     -243  A     -130
      A     -256  A     -62   A     86    A     -146  A     -187  A     -56   A     -55   A     -59   A
      -131  A     -154  A     -79   A     -76   A     -34   A     -95   A     -12   A     -21   A     -202
      A     -112  A     -118  A     -90   A     -137  A     -157  A     -172  A     -47   A     -62   A
      -58   A     -161  A     -48   A     -176  A
AFFX-BioB-M_at (endogenous control) AFFX-BioB-M_at    -200  A     -248  A     -153  A     -218  A     -177
      A     -249  A     -23   A     -36   A     -74   A     -187  A     -43   A     -44   A     -114  A
      -126  A     -136  A     -118  A     -98   A     -144  A     -118  A     -172  A     -13   A     -274
      A     -185  A     -142  A     -87   A     -51   A     -370  A     -122  A     -442  A     -198  A
      -217  A     -215  A     -531  A     -284  A
AFFX-BioB-3_at (endogenous control) AFFX-BioB-3_at    41    A     262   A     17    A     -163  A     -28
      A     -410  A     -7    A     -141  A     170   A     312   A     43    A     12    A     23    A
      -50   A     49    A     -30   A     -153  A     -17   A     59    A     12    A     8     A     59
      A     24    A     212   A     102   A     -82   A     -77   A     38    A     -21   A     -5    A
      63    A     -46   A     -124  A     -81   A
AFFX-BioC-5_at (endogenous control) AFFX-BioC-5_at    328   A     295   A     276   A     182   A     266
      A     24    A     142   A     252   A     174   A     142   A     177   A     129   A     146   A
      211   A     180   A     68    A     237   A     152   A     270   A     172   A     38    A     309
      A     170   A     314   A     319   P     178   A     340   A     31    A     396   A     141   A
      95    A     146   A     431   A     9     A
AFFX-BioC-3_at (endogenous control) AFFX-BioC-3_at    -224  A     -226  A     -211  A     -289  A     -170
      A     -535  A     -233  A     -201  A     -32   A     114   A     -116  A     -108  A     -171  A
      -206  A     -257  A     -110  A     -215  A     -174  A     -229  A     -137  A     -128  A     -456
      A     -197  A     -401  A     -283  A     -135  A     -438  A     -201  A     -351  A     -256  A
      -191  A     -172  A     -496  A     -294  A
AFFX-BioDn-5_at (endogenous control)      AFFX-BioDn-5_at   -427  A     -493  A     -250  A     -268  A
      -326  A     -810  A     -284  A     -384  A     -318  A     -148  A     -184  A     -301  A     -227
      A     -287  A     -273  A     -264  A     -122  A     -289  A     -383  A     -205  A     -245  A
      -581  A     -400  A     -452  A     -385  A     -320  A     -364  A     -226  A     -394  A     -206
      A     -230  A     -596  A     -696  A     -493  A
```

NOTE: This is only the 1st seven lines of this independent data set. The complete independent data set is 7130 lines in length.

table_ALL_AML_samples.txt

```
LEUKEMIA SAMPLES INFORMATION


Samples     ALL/AML        BM/PB    T/B-cell    FAB         Date/Gender        % Blasts   Treatment   PS
      Source
                                   (if ALL)       (if AML)                                 Response


INITIAL SET

1           ALL         BM    B-cell                        09/04/96 - M                              1.00
      DFCI
2           ALL         BM    T-cell                                 - M                              0.41
      DFCI
3           ALL         BM    T-cell                                 - M                              0.87
      DFCI
4           ALL         BM    B-cell                                                            0.91  DFCI
5           ALL         BM    B-cell                                                            0.89  DFCI
6           ALL         BM    T-cell                                 - M                              0.76
      DFCI
7           ALL         BM    B-cell                        03/25/83 - F                              0.78
      DFCI
8           ALL         BM    B-cell                                 - F                              0.77
      DFCI
9           ALL         BM    T-cell                                 - M                              0.89
      DFCI
10          ALL         BM    T-cell                        07/23/87 - M                              0.56
      DFCI
11          ALL         BM    T-cell                        06/25/85 - M                              0.74
      DFCI
12          ALL         BM    B-cell                        09/17/85 - F                              0.20
      DFCI
13          ALL         BM    B-cell                        07/27/88 - F                              1.00
      DFCI
14          ALL         BM    T-cell                        11/27/87 - M                              0.73
      DFCI
15          ALL         BM    B-cell                        03/25/89 - F                              0.98
      DFCI
```

```
16          ALL         BM    B-cell                      02/12/90 - M                          0.95
      DFCI
17          ALL         BM    B-cell                      09/26/90 - M                          0.49
      DFCI
18          ALL         BM    B-cell                               - F                          0.59
      DFCI
19          ALL         BM    B-cell                                                      0.80  DFCI
20          ALL         BM    B-cell                                                      0.90  DFCI
21          ALL         BM    B-cell                      01/24/84 - M                          0.76
      DFCI
22          ALL         BM    B-cell                      05/27/88 - M                          0.37
      DFCI
23          ALL         BM    T-cell                      07/09/91 - M                          0.77
      DFCI
24          ALL         BM    B-cell                      05/19/81 - M                          0.92
      DFCI
25          ALL         BM    B-cell                      02/18/82 - M                          0.43
      DFCI
26          ALL         BM    B-cell                               - F                          0.89
      DFCI
27          ALL         BM    B-cell                               - F                          0.82
      DFCI
28          AML         BM                M2                         79    Failure        0.44  CALGB
29          AML         BM                M2                         34    Failure        0.74  CALGB
30          AML         BM                M5                         93    Failure        0.80  CALGB
31          AML         BM                M4                         77    Failure        0.61  CALGB
32          AML         BM                M1                         86    Failure        0.47  CALGB
33          AML         BM                M2                         70    Failure        0.89  CALGB
34          AML         BM                M2                         77    Success  0.64  CALGB
35          AML         BM                M1                         67    Success  0.21  CALGB
36          AML         BM                M5                         76    Success  0.94  CALGB
37          AML         BM                M2                         44    Success  0.95  CALGB
38          AML         BM                M1                         80    Success  0.73  CALGB

INDEPENDENT SET

39          ALL         BM    B-cell                               - F                          0.78
      DFCI
40          ALL         BM    B-cell                      05/16/80 - F                          0.68
      DFCI
```

41	ALL	BM	B-cell			-	F			0.99	DFCI
42	ALL	BM	B-cell			-	F			0.42	DFCI
43	ALL	BM	B-cell			-	F			0.66	DFCI
44	ALL	BM	B-cell		11/19/98	-	F			0.97	DFCI
45	ALL	BM	B-cell		11/19/98	-	M			0.88	DFCI
46	ALL	BM	B-cell		01/08/99	-	F			0.84	DFCI
47	ALL	BM	B-cell		09/05/86	-	M			0.81	DFCI
48	ALL	BM	B-cell		02/28/92	-	F			0.94	DFCI
49	ALL	BM	B-cell			-	M			0.84	DFCI
50	AML	BM		M4				93	Failure	0.97	CALGB
51	AML	BM		M2				57	Failure	1.00	CALGB
52	AML	PB		M4				86	Success	0.61	CALGB
53	AML	BM		M2				76	Success	0.89	CALGB
54	AML	BM		M4		-	F			0.23	St-Jude
55	ALL	BM	B-cell			-	F			0.73	St-Jude
56	ALL	BM	B-cell			-	F			0.84	St-Jude
57	AML	BM		M2		-	F			0.22	St-Jude
58	AML	BM		M2						0.74	St-Jude
59	ALL	BM	B-cell			-	F			0.68	St-Jude
60	AML	BM		M2		-	M			0.06	St-Jude
61	AML	BM		M1						0.40	St-Jude
62	AML	PB				-	M			0.58	CCG
63	AML	PB				-	F			0.69	CCG
64	AML	PB				-	M			0.52	CCG
65	AML	BM				-	M			0.60	CCG
66	AML	BM				-	M			0.27	CCG

67		ALL	PB	T-cell	05/21/97 - M		0.15
	DFCI						
68		ALL	PB	B-cell	04/06/98 - M		0.80
	DFCI						
69		ALL	PB	B-cell	09/15/98 - M		0.85
	DFCI						
70		ALL	PB	B-cell	12/11/98 - F		0.73
	DFCI						
71		ALL	PB	B-cell	07/18/98	0.30	DFCI
72		ALL	PB	B-cell	07/28/98	0.77	DFCI