1997

A Comparison of pattern classification techniques for orienting chest X-rays

Martin R. Hoffmann

Rochester Institute of Technology
Computer Science Department

A Comparison of Pattern Classification Techniques for Orienting Chest X-rays

by
Martin R Hoffmann

A thesis, submitted to
The Faculty of the Computer Science Department,
in partial fulfillment of the requirements for the degree of
Master of Science in Computer Science.

Approved by:

_____
Professor Peter Anderson

_____
Dr. Roger Gaborski

_____
Professor John Biles

April 23, 1997

Permission To Copy

Title of thesis: A Comparison of Pattern Classification Techniques for Orienting Chest X-rays

I, Martin R. Hoffmann, hereby **grant permission** to the Wallace Library of the Rochester Institute of Technology to reproduce my thesis in whole or in part. Any reproduction will not be for commercial use or profit.

Date: 4/23/97 Signature of Author: _____

Abstract

The problem of orienting digital images of chest x-rays, which were captured at some multiple of 90 degrees from the true orientation, is a typical pattern classification problem. In this case, the solution to the problem must assign an instance of a digital image to one of four classes, where each class corresponds to one of the four possible orientations.

A large number of techniques are available for developing a pattern classifier. Some of these techniques are characterized by independent variables whose values are difficult to relate back to the problem being solved. If a technique is highly sensitive to the values of these variables, the lack of a rigorous way of defining them can be a significant disadvantage to the inexperienced researcher.

This thesis presents experiments by the author to solve the chest x-ray orientation problem using four different pattern classification techniques: genetic programming, an artificial neural network trained with back propagation, a probabilistic neural network, and a simple linear classifier. In addition, the author will demonstrate that an understanding of the design of a feature set may allow a programmer to develop a traditional program which does an adequate job of solving the classification problem. Comparisons of the different techniques will be based not only on their success at solving the problem, but also on the time required to find an acceptable solution and the degree to which each technique is sensitive to the values of the variables which characterize it.

The thesis demonstrates that all of the techniques can be used to derive very accurate chest x-ray orientation classifiers. While it is dangerous to generalize the results of these experiments to pattern classification problems in general, the author will argue that the magnitude of the differences in performance between the different techniques minimizes this danger. In particular, the experiments suggest that the linear classifier is so computationally inexpensive that it is always worth trying, unless there is a priori knowledge that it will fail. The experiments also suggest that genetic programming is much more computationally expensive than are the linear classifier, artificial neural network, and probabilistic neural network techniques.

Of the four conventional pattern classification techniques which were examined, it will be shown that the artificial neural network produced the most accurate classifiers for the x-ray orientation problem. In addition, the results of a number of trials suggest that the final accuracy of the classifier is relatively insensitive to the values of the parameters which characterize this technique, making it an appropriate choice for the inexperienced researcher. With respect to the ability of the resulting classifier to accurately orient sample x-rays which were not included in the training set, the artificial neural network performed well, when compared to the other techniques.

Although the classifiers produced by the genetic programming technique were significantly more expensive to construct and were slightly less accurate than the best artificial neural networks, the results of genetic programming experiments can provide insights into the problem being studied, which would be difficult to discern from the classifiers produced by the other techniques. For example, one of the classifiers which was produced by genetic programming uses only eight of the twenty feature values extracted from the sample x-ray. Not only does this reduce the cost of extracting the feature values from an unknown sample, but the classifier itself would be much more efficient to evaluate than the classifiers produced by any of the other techniques.

Table Of Contents

1. Introduction

In order to investigate the problem of chest x-ray orientation, the author implemented classifiers using four different pattern classification techniques: genetic programming, artificial neural networks trained with back propagation, probabilistic neural networks, and simple linear classifiers. In addition, the author employed knowledge used in the design of a particular feature set to develop a more traditional program which could also be used as a chest x-ray orientation classifier.

Each of these classifiers was implemented using Sun C++ 3.1, and all of the experiments were executed on a Sun SPARCstation 10 with 64 MBytes of RAM, running Solaris V2.3.

The author obtained 238 sample chest x-ray images from the Eastman Kodak Company Research Labs. Each sample was repeatedly rotated 90 degrees to produce a sample image in each of the four target orientations, resulting in a total of 952 sample images.

The original samples were 45x55 pixel, 8-bit gray-scale images. These were cropped about their centers to produce square, 45x45 pixel images. Two feature sets were extracted from the 952 samples images, to be used in training and evaluating the performance of the various techniques.

The simpler of the two feature sets was derived by summing pixel values across various paths through the image. A second more complex feature set was designed to detect the orientation of the dark region which appears between the lungs in typical chest x-ray images.

A number of experiments were performed in which a classifier was trained using a fraction of the sample images, and the overall accuracy of the resulting classifier was evaluated using the remaining samples. Care was taken to ensure that the samples used for training originated from a different set of the original 238 images than did the samples used for evaluating the final classifier.

Experiments using the genetic programming and artificial neural network techniques require the selection of values for a number of different independent variables which are difficult to relate to the problem being solved. The author employed the Latin Squares technique to develop a suite of experiments which tested the classifiers against various combinations of some of these settings.

This paper is organized as follows. Following this introduction are sections which describe each of the four pattern classification techniques which were used to solve the chest x-ray orientation problem. Each of these sections begins by briefly describing the technique, and concludes with a description of the author's implementation of the technique used in the experiments.

After the sections which describe the pattern classification techniques is a section which describes the sample x-ray images used in the experiments and the two feature sets which were extracted from these images. Subsequent sections of the paper will describe the experiments performed with each of the four techniques and two feature sets, including the setup of the experiments and the results from these experiments.

Finally, the author describes his attempt to write a traditional program for orienting the x-rays and offers his conclusions drawn from the results of the experiments described earlier.

2. Pattern Classification Techniques

This section of the paper describes the four different pattern classification techniques which were used to orient the chest x-rays. Each description begins with some background about the technique and ends with details of the implementation used in the experiments described in this paper.

2.1. Genetic Programming

In 1975, John Holland published *Adaptation In Natural And Artificial Systems*, which showed how the evolutionary process can be applied to problems in adaptation. The technique of Genetic Algorithms was developed as a means of using evolution to solve such problems and is the foundation upon which the Genetic Programming technique is based.

2.1.1. Genetic Algorithms

With the simplest form of Genetic Algorithm, candidate solutions to a problem are represented by fixed-length strings. A population of candidate solutions are randomly selected from the set of all possible solutions and the individuals in the population are ranked according to their ability to solve the target problem. Based on the results obtained for this population, a new population is drawn from the solution space and evaluated. This is repeated until an acceptable solution is found.

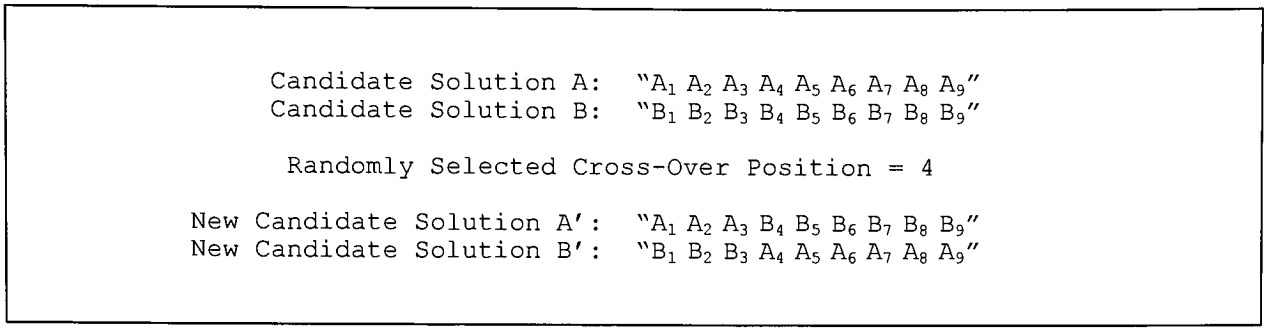
A new population of candidate solutions is generated from the previous population by applying three different methods of *breeding*: asexual reproduction, cross-over, and mutation. Asexual reproduction involves selecting a candidate from the current population and copying it into the new population.

Figure 1: Example Of Breeding By Asexual Reproduction

Candidate Solution A:	"A_1 A_2 A_3 A_4 A_5 A_6 A_7 A_8 A_9"
New Candidate Solution A':	"A_1 A_2 A_3 A_4 A_5 A_6 A_7 A_8 A_9" (same as original)

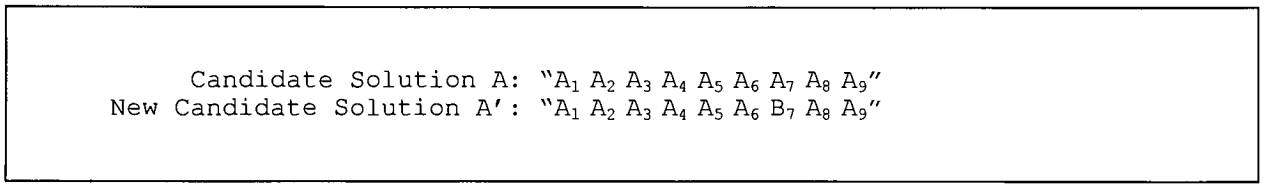
Cross-over is done by selecting two individuals from the existing population and randomly selecting a point where the strings representing the two solutions are crossed to produce two new candidate solutions.

Figure 2: Example Of Breeding By Cross-Over



Like asexual reproduction, mutation involves selecting a single individual from the current population and copying it into the new population. With mutation, however, one character in the new string is randomly selected and changed to a different value.

Figure 3: Example Of Breeding By Mutation



While mutation can preserve variation in the resulting population, in practice it sees little use (Koza, 1992:105). The relative frequencies by which asexual reproduction, cross-over, and mutation are used to generate the new population of candidate solutions are tunable parameters of the application of the algorithm.

By relating the fitness of a candidate solution to the probability that it is selected for breeding, the Genetic Algorithm directs the search for the optimal solution to the target problem. Selection is done with replacement, so that the same solution may be selected to participate in a number of breeding operations. Holland showed that the generation of the new population using these techniques was nearly optimal in minimizing the cost associated with sampling the solution space (Holland, 1992:139).

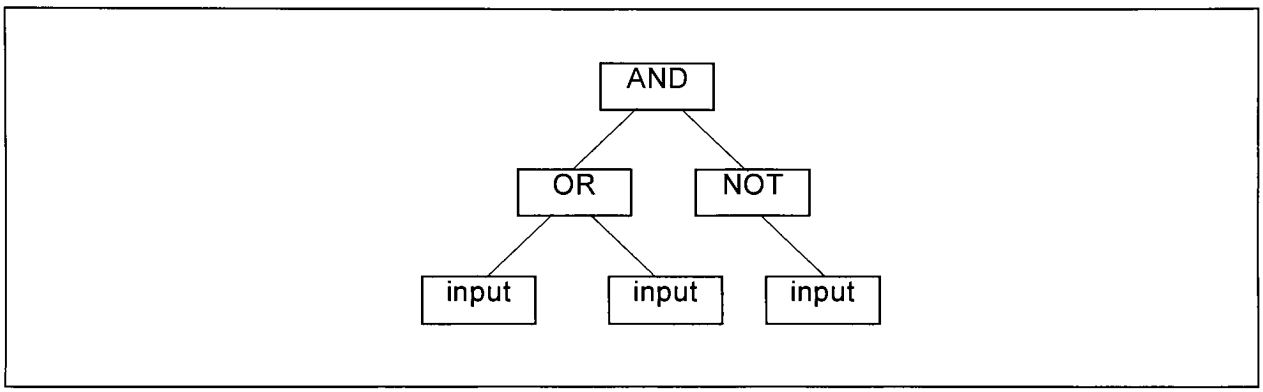
2.1.2. Genetic Programming

John Koza has developed a variation on Genetic Algorithms called Genetic Programming. With Genetic Programming, solutions to the target problem are represented by programs rather than simple strings (actually, candidate solutions are stored as parse trees).

In his book, *Genetic Programming*, Koza argues that representing candidate solutions as parse trees instead of strings increases the expressive power of the algorithm, without invalidating the work of Holland. Therefore, he concludes that the algorithm described below is a near optimal method of sampling a solution space consisting of candidate programs (Koza, 1992:116).

Each node within a candidate parse tree represents an input or function. Terminal nodes are inputs or functions which require no arguments. Non-terminal nodes represent functions which take one or more arguments. The values of these arguments are found by evaluating child nodes in the tree.

Figure 4: Example Of A Genetic Programming Parse Tree



In order to ensure that randomly generated parse trees and parse trees resulting from breeding are always valid programs, Genetic Programming requires that the program inputs, function arguments, and function return values be of the same data type.

In a Genetic Programming experiment, a population of programs is randomly generated. Each member of the population is evaluated against test data and the programs are ranked as to their fitness for solving the target problem. A new population is generated by breeding individuals from the previous population.

The terminal and non-terminal nodes available for program generation depend on the nature of the problem being solved, and are part of the characterization of a particular experiment. The fitness measure used to associate a numerical rating for the suitability of each candidate program in solving the target problem is also domain specific.

For example, in the experiments conducted by the author, each x-ray was reduced to a set of 20 feature values. The set of terminal nodes used in the Genetic Programming experiments included one node for each of these 20 values,

plus one node for each of four integer constants: 0, 1, 2, and 3. The set of non-terminal nodes included addition, the greater-than comparison operator, a conditional "if" operator, and bitwise AND, OR, and NOT operators.

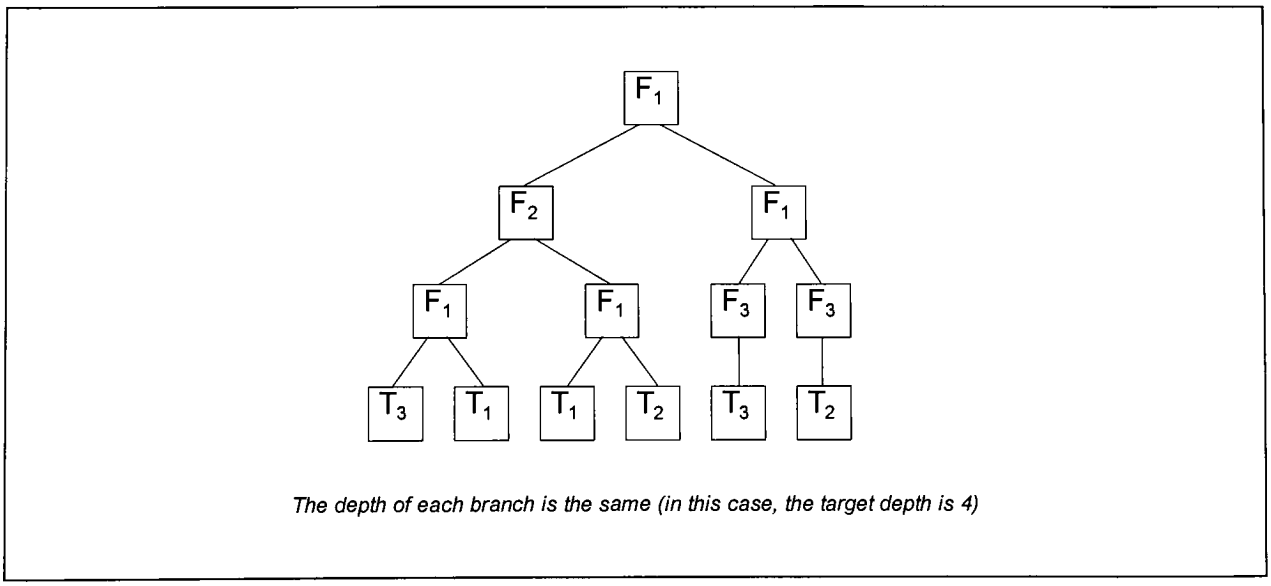
The fitness of a particular solution was determine by evaluating the candidate program repeatedly against the feature vectors associated with a fixed set of sample x-ray images. The low order two bits of the result of an evaluation of the program was used to select from amongst the four possible orientation values. The fitness value assigned to the program was simply the fraction of correctly classified training samples.

2.1.3. Creating The Initial Population Of Programs

The initial population of candidate programs is generated randomly using the sets of terminal and non-terminal nodes defined for the experiment. Koza describes two methods of generating different shaped parse trees and then recommends a hybrid method called "ramped half-and-half" (Koza, 1992:91).

The first of the two basic parse tree generation methods is the *full* method. The *full* method begins by selecting a target depth for the parse tree. The tree is generated from the top down. When a node is needed as an argument to a function in the previous level of the tree, a non-terminal node is randomly selected, if and only if the depth of the current branch of the tree is less than the target depth. If the depth of the current branch is equal to the target depth, a terminal node is randomly selected. Uniform probabilities are used for node selection.

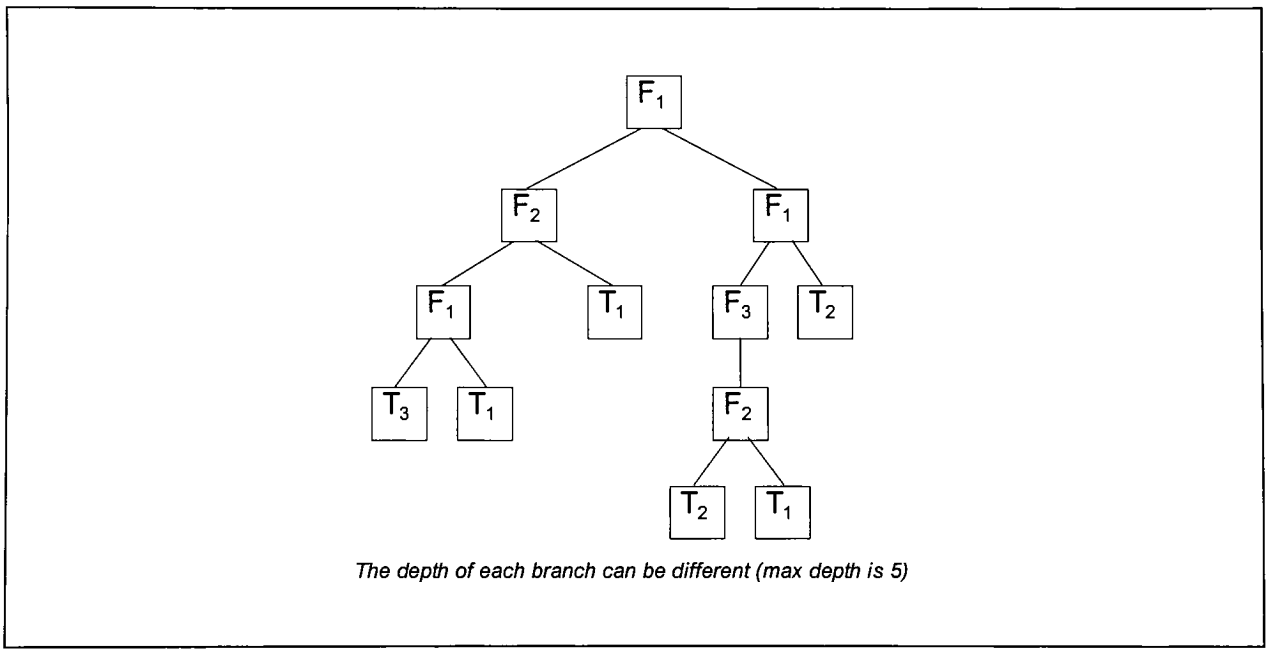
Figure 5: A Parse Tree Generated By The "Full" Method



The second of the two basic methods of parse tree generation is the *grow* method. With the *grow* method, only the maximum depth of all branches is pre-defined. When a node is needed as an argument to a function in the previous level of the tree and the depth of the current branch is less than the target maximum depth, then a node is randomly selected from either the set of terminal or non-terminal nodes. This means that the lengths of the different branches of the parse tree can vary.

Koza suggests that the selection of nodes at intermediate levels of the tree be made uniformly from the union of the sets of terminal and non-terminal nodes (Koza, 1992:92). In practice, this leads to rather uninteresting parse trees when the number of terminal nodes is significantly greater than the number of non-terminal nodes.

Figure 6: A Parse Tree Generated By The "Grow" Method



The *ramped half-and-half* method is a hybrid of these two methods of parse tree generation. With this method, half of the initial population of parse trees is generated using the *full* method and the other half is generated using the *grow* method. The minimum target depth used to generate any tree is two. The maximum target depth is a tunable parameter typically around six (Koza, 1992:116). The same number of parse trees are generated for each target depth, using each of the two basic tree generation methods.

During the generation of the initial population of programs, duplicate parse trees are eliminated by replacing one of the trees with a new tree with the same characteristics. Because the programs do not interact with one another, there is no advantage to having duplicate programs in the initial population of a Genetic Programming experiment.

After the first generation of programs is created, the programs are evaluated against test data and are assigned fitness values which define their relative success at solving the target problem. Subsequent generations of programs are created by breeding individuals from the current generation.

2.1.4. Creating Subsequent Generations Of Programs

As with the general Genetic Algorithm, breeding is done by asexual reproduction, cross-over, and mutation. In this case, cross-over is performed by randomly selecting a node in each of the parent programs. The sub-trees rooted at these two nodes are then swapped to generate two programs for the new population (see Figure 7).

The mutation operation (which is rarely used and was not used in the experiments described in this paper) is performed by pruning a randomly selected sub-tree of a selected program and replacing it with another randomly generated sub-tree.

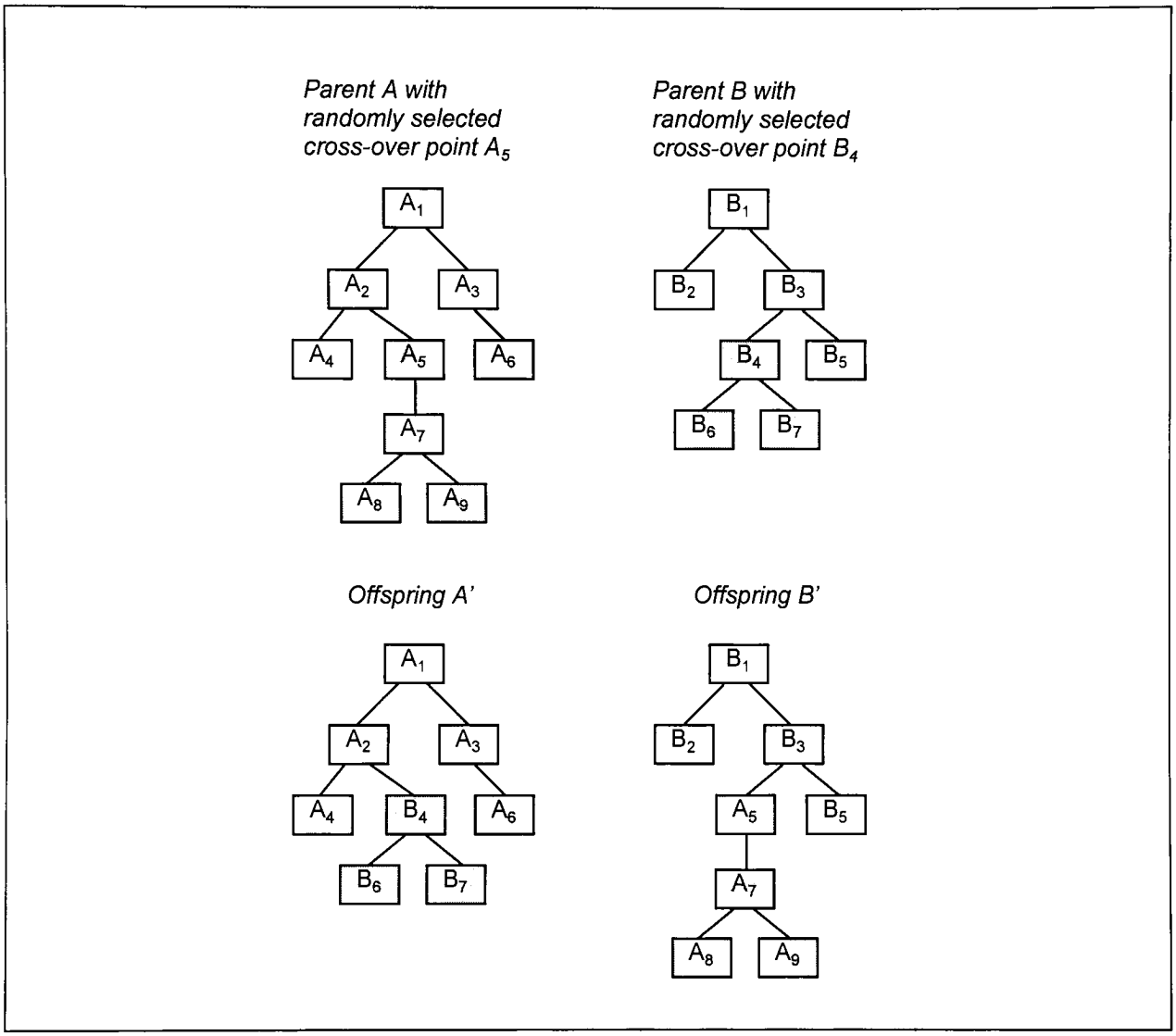
In addition to asexual reproduction, cross-over, and mutation Koza describes three methods of generating new programs from an existing population of programs (Koza, 1992:107-112). Although described below, these methods were not employed by the author in his experiments.

A *permutation* operation generates a new program by randomly re-ordering the arguments to one of the functions in an existing parse tree. The *editing* operation modifies an existing parse tree by recursively applying a set of domain independent and optionally domain dependent editing rules. Koza gives an example of a domain independent editing rule as follows:

> "If any function that has no side effects and is not context dependent has only constant atoms as arguments, the editing operation will evaluate the function and replace it with the value obtained from the evaluation" (Koza, 1992:108)

The final method of generating new programs is *encapsulation*. With encapsulation, a randomly selected sub-tree of a randomly selected candidate program is wrapped by a new primitive, which then replaces the sub-tree in the original program. The new primitive is like a subroutine. If mutation is being used to breed programs, the new primitive can be used as a terminal node in the randomly generated parse-trees created by the mutation operation.

Figure 7: Example Of Cross-Over In Genetic Programming



As with Genetic Algorithms, the relative frequencies with which the different breeding methods are employed in generating a new population of programs are tunable parameters.

Programs are selected for breeding with a frequency which is proportional to their relative fitness at solving the target problem. The same program can be selected to participate in multiple breeding operations, and because asexual reproduction is also used during breeding, the same program can appear more than once in the new population.

In his book, Koza describes other techniques which are useful for improving the rate at which a Genetic Programming experiment converges to a solution. The author of this paper included two of these techniques in his implementation: *decimation* and *greedy over-selection.*

Because the generation of the initial population of programs is almost purely random, there are likely to be a large number of candidate programs in this first generation with extremely poor fitness ratings. *Decimation* provides a fast way of eliminating the poorest candidate programs, by simply removing a fixed number of them from the initial population, before breeding begins.

Even after employing decimation to trim the initial population of an experiment, the number of programs which remain may be quite large. The probability that even the fittest candidate program is selected for breeding may be relatively small. *Greedy over-selection* is a method which increases the probability that the programs which are better candidate solutions are selected for breeding.

With greedy over-selection, the programs from the current population are divided into two groups. The first group contains the fittest individuals which collectively account for some total fraction of overall fitness of the population. The second group contains all of the other programs. When a program needs to be selected for breeding, a group is randomly selected and then a program is randomly selected from amongst the candidate programs in the group.

The probabilities of selecting between the two groups are skewed to greatly favor the group of programs which contains the fittest individuals. Once a group is selected, a program is selected based on the relative fitness of the programs within the group. Koza's rules of thumb are that the first group is selected 80% of the time and that the fraction of total fitness which determines the division of programs between the two groups depends on the total number of programs as follows (Koza, 1992:99):

Table 1: Koza's Allocation Of Programs For Greedy Over-Selection

Number Of Programs	Fraction Of Total Fitness From Group-1 Programs
1,000	32%
2,000	16%
4,000	8%
8,000	4%

2.1.5. Characteristics Of A Genetic Programming Experiment

With the inclusion of decimation and greedy over-selection, the set of tunable parameters associated with a genetic programming experiment includes:

Table 2: Tunable Parameters For Genetic Programming Experiments

Terminal Set (the set of program inputs which can appear as parse tree leaves) Function Set (the set of functions which can appear as nodes in the parse trees) Fitness Measure (a method of assigning a numeric rating to the fitness of a program)
Population Size (number of programs in the initial population) Maximum Number Of Generations (number of iterations during the experiment) Maximum Depth Of Parse Tree In Initial Population Maximum Allowable Depth Of Parse Tree During Experiment
Probability Of Cross-Over Probability Of Asexual Reproduction Probability Of Mutation Probability Of Permutation Probability Of Edit Probability Of Encapsulation
Probability Of Selecting Leaf Node During Cross-Over
Fraction Of Programs Discarded By Decimation Step Fraction Of Overall Fitness Allocated To Greedy Over-Selection Group-I Probability Of Selecting Program From Greedy Over-Selection Group-I

Koza claims that the accuracy of the Genetic Programming algorithm is relatively insensitive to many of these variables and he typically uses the same set of values for most of his experiments (Koza, 1992:114). In the experiments described later, the author varies a small number of these parameters to evaluate the sensitivity of the chest x-ray orientation problem to these values.

2.1.6. Implementing The Genetic Programming Simulator

The implementation of the Genetic Programming algorithm by Koza was done using LISP, because of the relative ease with which individual programs, stored as s-expressions, could be manipulated. To perform the Genetic Programming experiments described in this paper, the author developed a reasonably flexible implementation of the algorithm using the C++ programming language.

Unlike LISP, C++ is a strongly typed language. To make the Genetic Programming environment as flexible as possible, the classes described below are in general implemented as template classes, parameterized by the type of data associated with the program inputs, function arguments, and function return values of the generated parse trees (there is one data type for all of these). For clarity in the descriptions which follow, the author will forego the C++ template notation in references to class names after the first (e.g., *Node<T>* and *Node* will be used to refer to the same class).

The most basic of classes in the design is the *Node<T>* class. *Node* is an abstract base class for all terminal and non-terminal nodes which can appear in a parse tree. Derived classes are implemented for each type of non-terminal node to be included in the experiment, and a single class, *TerminalNode<T>*, is used to represent the leaf nodes of the parse trees.

Each derived class implements a small number of member functions which characterize the type of operation performed by this class of node. These functions include the following functions which are declared pure virtual in *Node*:

int args (void);
: Returns the number of arguments required as input to this node. *TerminalNode* implements this function to return zero. The integer add node described later returns the value 2, because it requires two inputs: the left and right addends.

*const char *name (void);*
: Returns a short descriptive name for the class of node, which is used when a parse tree is displayed to the user.

*Node *gnu (void);*
: Returns a new instance of the same exact class as the receiver of the gnu( ) call. This is used to generate new nodes of a particular class at run time.

In the author's original design for the simulator, each derivative *Node* also implemented a virtual function, eval( ), which obtained the node's inputs from its child nodes in the parse tree and combined these inputs to produce the result for the node. For example:

```
int IntAdd::eval (void)
{
   return ( child[0]->eval( ) + child[1]->eval( ) );
}
```

The inputs to the program were stored in an array, and each instance of *TerminalNode* was assigned an index into the array. The implementation of eval( ) in *TerminalNode* simply returned the appropriate element of the input array.

Because of the recursive calls to eval( ) in its implementation by non-terminal nodes, an entire parse tree could be evaluated by calling eval( ) against the root node of the tree. While the author believes this to be good object-oriented design, the evaluation of a parse tree then requires a number of virtual function calls, equal to the number of nodes in the parse tree. Virtual function calls are relatively expensive when compared to the execution of a switch statement, upon which a parse tree interpreter might be implemented. In fact, through a separate experiment, the author estimated that a virtual function call of the form shown above requires about twice as much CPU time as the execution of a switch statement (in the environment in which the experiments were conducted).

In order to avoid introducing an unfair bias against the Genetic Programming technique, the author implemented a different approach to program evaluation. In this approach, a 4-byte code is associated with each class of non-terminal node and each unique input value. For non-terminal nodes, the high-order bit of the code is set to distinguish the code from that of a terminal node. For terminal nodes, this bit is not set and the code is equal to the node's index into the input array.

Through a process, which will be referred to as *flattening*, a parse tree is compiled into a one dimensional array of these byte codes. Flattening is accomplished by calling a virtual flatten( ) function against the root node of the tree. This function is passed a pointer to an array and a reference to an index variable which points to the next writable position in the array. Each node responds to flatten( ) by calling flatten( ) against each child, and then pushing the byte code which corresponds to the node itself, onto the end of the array:

```
void IntAdd::flatten (unsigned int *array, unsigned int &index)
{
   child[1]->flatten (array, index);
   child[0]->flatten (array, index);
   array[index++] = IntAdd_Byte_Code;
}
```

In order to determine the size of the array required to hold the flattened version of a particular parse tree, *Node* also implements a virtual function to return the number of bytes required to flatten the sub-tree rooted at a node. The default implementation of this function, in class *Node*, is sufficient for most derived classes. That implementation

simply calls the function recursively against each child (summing the results) and then adds one, to account for the byte-code of the node itself.

Thus, the flattening process requires two calls to virtual functions for each node in the parse tree. Fortunately, the flattened parse tree will be evaluated a large number of times, resulting in a net savings of CPU time (in each of the Genetic Programming experiments described later in this paper, the original implementation of the simulator was tested and was found to be at least twice as slow as this new version, whose results are used for comparison with the other classification techniques).

To evaluate the flattened parse tree, a *Stack<T>* class is needed to hold intermediate results. The stack provides an *inlined* member function to push a value on to the stack, and another inlined function which pops the stack and returns the value just removed. Starting with an empty stack, the evaluator function iterates through the array of byte-codes. If the current byte-code is that of a terminal node, then the corresponding value from the input array is pushed on to the stack. Otherwise, an inlined eval( ) function is called against the class of *Node* indicated by the byte-code. This eval( ) function pops the required arguments from the stack, computes the result of the operation, and pushes the result onto the stack. After the last element of the array has been processed, the stack contains a single element which corresponds to the program result.

```
int  evaluateByteCodes (unsigned int *array, unsigned int num_codes)
{
   static Stack<int> s;
   s.clear( );

   for (int i=0; i<num_codes; i++)
      switch (array[i])
      {
      case IntAdd_Byte_Code:   IntAdd::eval(s);   break;
      ...
      default: s.push( TerminalNode<int>inputs[array[i]] );
      }

   return s.pop( );
}


inline void IntAdd::eval (Stack<int> &s)
{
   s.push (  s.pop( )  +  s.pop( )  );
}
```

This design has the disadvantage that a hard-coded switch statement must be formulated in the implementation of evaluateByteCodes( ), based on the set of non-terminal node types to be included in the experiment. Unfortunately,

the design becomes even more obfuscated, with the introduction of the "conditional if" operation as a type of non-terminal node. The if-operator accepts three arguments. If the first argument evaluates to a non-zero value, then the operation returns its second argument. Otherwise, the operation returns its third argument (thus the if-operator is equivalent to the C++ "? :" operator). This operation could be implemented as follows:

```
inline void IntIf::eval (Stack<int> &s)
{
   int arg1 = s.pop( );
   int arg2 = s.pop( );
   int arg3 = s.pop( );

   return (arg1 ? arg2 : arg3);
}
```

But this design is horribly inefficient, since all three arguments are evaluated, even though only one of the second and third is actually needed (in the author's experiments, there is no chance of side-effects being introduced by the evaluation of a sub-tree, so the evaluation need not be done at all, if the result is not used). In order to prevent the unnecessary evaluation of unused results, a way is needed to delay the evaluation of the latter two arguments to the if-operator until a decision is made as to which will be evaluated and which will be ignored.

The author's solution is similar to one which he later discovered was described by Keith and Martin (Keith, 1994:294). First, a new byte-code called a "skip" code is introduced, which is composed of an identifying flag and an array index value which represents the element to which the evaluator should proceed next. This new skip code is used in the implementation of flatten for the class *IntIf*:

```
void IntIf::flatten (unsigned int *array, unsigned int &index)
{
   child[0]->flatten (array, index);    // encode 1st arg
   array[index++] = IntIf_Byte_Code;    // add self to byte-code array

   unsigned int save = index++;         // leave room for "goto"
   child[1]->flatten (array, index);    // encode 2nd arg

   array[save] = SkipFlag | (index+1); // "goto" points past 2nd arg
   save = index++;                      // leave room for 2nd "goto"

   child[2]->flatten (array, index);    // encode 3rd arg
   array[save] = SkipFlag | index;      // "goto" points past 3rd arg
}
```

The flattened version of an *IntIf* operation requires two additional slots for the skip codes. This is the main reason that the function which determines the number of array elements required by a flattened parse tree must be a virtual

function. Unlike other *Node* eval( ) functions, the implementation of eval( ) for *IntIf* does not push a result value onto the stack. Instead, it adjusts the index which is being used to walk through the array of byte-codes during program evaluation:

```
inline void  IntIf::eval (Stack<int> &s, int &i)
{
   if ( s.pop( ) ) i++;
}
```

This function pops the stack to retrieve the first argument to the if-operator. If this argument is zero, then the eval( ) function does nothing. The next byte-code processed will be a skip code which causes the evaluator to jump past the byte-codes associated with the if-operator's second argument to the start of byte-codes associated with the if-operator's third argument. Subsequently, when that section of the byte-code array has been processed, the value corresponding to the third argument will reside at the top of the stack, as if it had been returned by the if-operator.

However, if the first argument to the if-operator is non-zero, then the eval( ) function increments the index being used by the evaluator, so that it will miss the skip code. At this point, the second argument to the if-operator will be evaluated and pushed onto the stack. The byte-code which immediately follows the flattened second argument to the if-operator is another skip code, which causes the evaluator to skip past the third argument, which does not need to be evaluated. Thus, in either case, only one of the second and third arguments to the if-operator will be evaluated.

In addition to the overrides of the pure virtual functions of class *Node* and the member functions related to program evaluation, each derived class which represents a non-terminal node implements a special constructor which is used only once, to register the class of non-terminal node in a static registry. The contents of this registry determines which Node derivatives will be created and used during the Genetic Programming experiment. A single static instance of the class is constructed using this constructor, to accomplish the registration. Here is an example of the complete implementation of the integer addition node:

```
const unsigned int IntAdd_Byte_Code = NonTerminalFlag | 0x00000001;

class IntAdd : public Node<int>
{
   private:

      static IntAdd a;  // static instance used to register class
      IntAdd (NodeType t) : Node<int>(t) { }  // for registration

   public:

      static void eval (Stack<int> &s)
      {
         s.push ( s.pop( ) + s.pop( ) );
      }

      int args (void) { return 2; }

      const char *name (void) { return "+"; }

      Node<int> *gnu (void) { return new IntAdd (instance); }

      unsigned int byte_code (void) { return IntAdd_Byte_Code; }

      void flatten (unsigned int *array, unsigned int &index)
      {
         child[1]->flatten (array, index);
         child[0]->flatten (array, index);
         array[index++] = IntAdd_Byte_Code;
      }
};

IntAdd IntAdd::instance (NONTERMINAL);
```

When the object module for IntAdd is linked into an executable, the static instance of the class is automatically registered at application startup. Registration involves storing a pointer to the instance in a static array of pointers maintained by the *Node* class. At run time, a class of non-terminal node is randomly selected by choosing an instance from this array. A new node of this class is produced by calling the gnu( ) function against that instance.

The registration of terminal nodes is handled a little differently, because there is only one C++ class to represent all of the terminal nodes in the experiment. At application startup, a global function named numberOfInputs( ) is called to determine the number of unique terminal nodes. This function must be provided by the experimenter. The value returned by this function is used to create the correct number of instances of *TerminalNode* (one per input). These instances are stored in another static array of *Node* pointers associated with the base class. At run time, when a new terminal node is needed, one of the instances maintained by the *Node* class is selected at random and cloned using the gnu( ) function implemented in *TerminalNode*.

Each instance of *TerminalNode* maintains an index into a static array of input values (the size of this array is also equal to the value returned by numberOfInputs( )). When a *TerminalNode* is created via gnu( ), the index value is copied so that the new node points to the same input value. During the run, the array of input values is loaded appropriately before a program is evaluated.

The *Node* base class maintains information about the node's place within its parse tree. This information includes:

- A pointer to the node's parent in the parse tree
- An array of pointers to the node's children in the parse tree
- The maximum depth of any sub-tree rooted at this node
- The number of non-terminal descendants of this node
- The number of terminal descendants of this node

This information is used during the construction and breeding of parse trees through a number of helper functions on the *Node* class (these functions describe the characteristics of the sub-tree rooted at the node and can return specific elements of that sub-tree given a depth-first numeric index into the sub-tree). The class which uses these member functions is the class whose instances represent the individual parse trees in the experiment, *Program<T>*.

Each *Program* stores a pointer to the root node of the parse tree it represents. The first time a *Program* is asked to evaluate itself, it creates the flattened version of the parse tree by calling the flatten( ) function against the root node of the tree. The *Program* then calls the global function, evaluateByteCodes( ), which was described earlier. The flattened version of the parse tree is not discarded until the Program is deleted. Thus, it can be evaluated multiple times.

In addition to storing a pointer to the parse tree and the byte-code array, *Program* maintains a number of other data members including data members for:

- the number of times the program has been run (i.e., evaluated).
- the cumulative fitness assigned to the program for all runs thus far.

- the number of *hits* scored by the program (a "hit" is scored each time the program returns an exactly correct result and is useful if some non-zero amount of the fitness measure is awarded for answers which are "close but not exact").

- the probability that this program should be selected for breeding.

- the number of times the program has actually been selected for breeding

- a description of how the program was produced (e.g., from cross-over) with data to identify the *parent* program(s) which were bred to produce this program

Much of this data is simply gathered for measuring statistics of the experiment, although the fitness measure is used to compute the selection probability and the selection probability is used during breeding to select programs at frequencies which are proportional to their fitness at solving the target problem.

The *Program* class provides three methods for creating a new program. During the creation of the first generation of programs within an experiment, a public constructor of *Program* is used to create new programs by either the "full" or "grow" methods described earlier. Subsequently, a copy constructor is provided for asexual reproduction and a breed( ) method is implemented for cross-over (mutation is also supported via a mutate( ) method, but the author has not tested this).

Both the "full" and "grow" methods of generating a tree from scratch are implemented by *Program* via recursive calls to its generateTree( ) member function. This function is passed an enumerated value to distinguish the method of generation and a pair of values which represent the minimum and maximum depths allowed for the generated tree.

The generateTree( ) function randomly selects a target depth, *d*, for the new tree, such that *d* lies in the range between the specified minimum and maximum allowable. The C++ run-time library function erand48( ) is used for this and all other situations in which random numbers are needed. Separate random number streams are used for each decision, by maintaining multiple seed arrays. The "full" method of tree generation is implemented as follows:

```
if  ( d is less than or equal to one )
{
   return a new, randomly selected, terminal node
}
else
{
   create a new, randomly selected, non-terminal node n

   for each argument required by n, generate a new sub-tree
   by calling generateTree( ) recursively, with minimum and
   maximum depth parameters both equal to (d - 1) (and using
   the "full" method)

   attach these arguments (i.e., sub-trees) to node n and return n.
}
```

This algorithm results in a tree in which the depth of each branch extending from the root is exactly equal to the depth selected in the original call to generateTree( ). *Program* implements the "grow" method of tree generation slightly differently than as described by Koza. In this implementation, there will always be exactly one branch which reaches the target maximum depth for the tree. The generateTree( ) method implements "grow" as follows:

```
if  ( d is less than or equal to one )
{
   return a new, randomly selected, terminal node
}
else
{
   create a new, randomly selected, non-terminal node n

   randomly select one of the arguments to n and create a
   sub-tree for it by calling generateTree( ) recursively,
   with minimum and maximum depths set to (d - 1) (and using
   the "grow" method)

   create the other arguments to n by calling generateTree( )
   recursively with a minimum depth parameter of 1 and a
   maximum depth of (d - 1) (again, the "grow" method is used
   in these recursive calls).

   attach these arguments to node n and return n.
}
```

This algorithm also differs from that described by Koza in that the depths of the various branches of the trees are independent of the relative difference in the number of classes of non-terminal nodes and the number of different inputs (i.e., terminal nodes). Thus the generated trees will not be artificially flattened when the number of inputs is significantly greater than the number of classes of non-terminal nodes included in the experiment.

The "full" and "grow" methods are only used to generate the initial, random population of programs evaluated as part of the first generation of the Genetic Programming experiments. Later generations of programs are created by breeding programs from the previous generation. The two methods of breeding employed in the experiments by the author are implemented via the copy constructor of the *Program* class and *Program*'s breed( ) method.

The copy constructor is used for asexual reproduction of a *Program* during breeding. The counts for number of runs, hits, and cumulative fitness are preserved in the new *Program*, but other data members, like the probability of selection for breeding and the number of times the program was selected are not copied from the existing *Program*, because these data members will depend on the results found in evaluating the other programs in the new generation.

As an optimization, the fitness of a program produced via asexual reproduction is not recomputed for the new generation of programs. Such evaluation would be a waste of time, since it is assumed that the fitness of a particular program is independent of generation in which it appears. Obviously, this same optimization cannot be applied to programs produced by cross-over, since the resulting programs will almost certainly be different than their parent programs.

Cross-over is implemented in the breed( ) method of class *Program*. The method is passed a pointer to another program (which serves as the second parent in the cross-over) and it returns the two programs which result from the cross-over. Cross-over is implemented as follows:

1. Use the copy constructor of Program to produce two new programs which are identical in structure to the two parent programs involved in the cross-over.

2. Randomly select a node from each of the two parse trees as the cross-over points.

3. Swap the sub-trees rooted at these two nodes by extracting each from its current parse tree and inserting it in the other parse tree.

Even if both of the parent programs were in fact the same instance of class *Program*, the above algorithm works, because the children are first produced by cloning, before the cross-over occurs. One situation which does require special handling is when one or both of the cross-over points are the root nodes of the child programs. In this case, an entire program is being replaced and the pointer stored to the root node in the *Program* object must be updated.

The selection of a cross-over point is a two step process. First, a decision is made as to whether the cross-over point should be a terminal or non-terminal node. The probability of selecting a terminal node in this step is a tunable parameter. Next the root *Node* is asked for the number of nodes of the selected type which exist in the tree rooted at that node. An index to one of these nodes is randomly generated and the root Node is asked to return a pointer to the node which corresponds to this index, given a depth first ordering of terminal or non-terminal nodes. The node which is returned becomes the cross-over point.

The two *Program* constructors and the breed( ) member function each implement a single step in the process of generating a population of test programs. These steps must be repeated an appropriate number of times to produce the complete population. The responsibility for generating and evaluating a population of programs falls to the *Generation<T>* class.

A *Generation* maintains a collection of *Program* objects. The class provides two constructors for creating new generations, a method to evaluate all of programs in the generation, and a method to save information about the generation to a file. This information includes the programs themselves, as well as some performance statistics, which will be described shortly.

The class *Generation* provides two constructors. The first constructor is the void constructor, which is used to create the first generation of programs for an experiment. This constructor implements the "ramped half-and-half" method of program generation described earlier, using the "full" and "grow" methods of parse-tree generation implemented by *Program*.

The second *Generation* constructor is passed a pointer to an existing generation. This is not a copy constructor. Instead, a new generation is produced by breeding the programs from the specified generation. The programs in the existing generation must already have been evaluated and sorted according to their relative fitness. In addition, the probability of selection for breeding must already have been computed for each program and a cumulative value for this probability stored in each of the programs. Breeding proceeds as follows:

1. A method of breeding is selected at random. The frequency with which each of the methods is selected is a tunable parameter of the experiment.

2. If the selected method is cross-over, then two programs from the previous generation are selected at random for breeding. Otherwise, a single program is selected. The probability that a program is

selected for breeding is proportional to its relative fitness within the previous generation of programs. Selection is done "with replacement".

3. The new program is produced (two programs in the case of cross-over) by calling the appropriate method of the *Program* class.

4. Steps 1 through 3 are repeated until a complete population of programs is generated.

Currently, only two methods of breeding are supported by the class *Generation*: cross-over and asexual reproduction. A method is selected by comparing a random number in the range 0.0 to 1.0 with the target probability for cross-over. If this random number is less than the probability of cross-over, then the method selected is cross-over.

The algorithm to select programs with appropriate selection frequencies is implemented as follows: a random number is generated in the range 0.0 to 1.0. The sorted collection of programs are searched in order of decreasing fitness, until a program is found whose cumulative probability of selection exceeds the random number. This program is selected. Greedy over-selection is implemented during the assignment of selection probabilities, so the division of the programs into two groups, as described for that technique, is implicit in the selection algorithm described here.

Once the entire population of programs has been created, the *Generation* can be evaluated by calling its eval( ) method. Although the process of evaluating the fitness of each program in the generation is implemented in a generic way, it depends on calls to a number of functions which are external to the class *Generation* and are provided by the experimenter to tailor the algorithm to the problem being studied. These functions include:

int numberOfInputs (void)

This function, which was mentioned earlier, defines the number of different terminal nodes to be registered at application startup.

int numberOfTests (void)

The evaluation of a program is broken down into one or more "tests". At the start of a test, the experimenter is asked to load the array of terminal node values with inputs appropriate to the next test. Then, the experimenter is asked to evaluate each program in the population, one at a time, and assign a fitness value to the program for this test.

The function numberOfTests( ) is called to determine the number of different tests which will be performed. This number must be independent of the generation being evaluated, because

programs produced by asexual reproduction are not reevaluated.

In the experiments described in this paper, a "test" consists of evaluating each program against the feature set extracted from a single sample x-ray image. The number of tests is therefore equal to the number of training samples.

The author's Genetic Programming framework actually evaluates the entire population of programs against only a subset of the tests indicated by numberOfTests( ). Then, after a selection probability has been assigned to each program, the remaining tests are used to evaluate the "best of generation" program. Since these latter tests never affect the selection of programs for breeding, they serve as a good measure of the success of the algorithm at finding a solution which is general enough to work well within the problem space represented by the test samples.

void loadInputs (void)

This function is called at the beginning of each test, to prepare the array of terminal node values for the start of the next test. In the experiments by the author, the set of inputs was exactly equal to the feature values extracted from a single sample x-ray (in one of its orientations), plus some constants. The implementation of loadInputs would populate the input array with values for a single training sample.

The number of calls to loadInputs( ) for each generation is equal to the number returned by numberOfTests( ). The version of loadInputs( ) implemented by the author automatically restarts at the beginning of the collection of training samples, after the features for the last training sample have been loaded. The function correctly presumes that this restart occurs only when a new *Generation* begins evaluating its programs.

*void evalProgram (Program<T> *program, double &fitness, int &hits, ...)*

The *Generation* class calls evalProgram( ) to obtain a fitness value for a specified program during the current test. The fitness values of a single program are summed over all tests to produce the program's cumulative fitness, which is a measure of the program's success at solving the target problem.

In its simplest form, evaluating a program during a test involves calling the eval( ) function of the Program and using the return value to assign a fitness value to the program for this test. Koza gives examples where the evaluation of a program for a single test may involve multiple calls to its eval( ) method, where each evaluation is used to adjusts the input array before the next call (Koza, 1992:147).

In these cases, the fitness measure is often inversely proportional to the number of times the program needs to be evaluated to satisfy some termination condition. This kind of test can be implemented using this author's framework, provided that the evalProgram( ) function restores the input array to its original state before returning.

double adjustFitness (double fitness)

This function is called once per program after it has been evaluated against all tests. The fitness value passed to this function is the cumulative fitness of the program determined over the course of the tests. The experimenter can implement
adjustFitness( ) to apply some domain specific adjustment to the fitness measure.

Because programs produced by asexual reproduction are not reevaluated, and their cumulative fitness is not readjusted, the adjustment performed by this function must be independent of both the characteristics of the current generation and the programs it contains. This restriction severely limits what can be done by this function, and in practice, the author found no need to modify the cumulative fitness values.

double perfectScore (void)

This function returns the adjusted fitness value that would be achieved by a program which performed perfectly during all of the tests. One of the stopping criteria for the experiment is finding a program whose adjusted fitness equals perfectScore( ).

*int evaluateByteCodes (unsigned int *array, unsigned int num_codes)*

This is the interpreter function described earlier. The actual return type depends on the type of data associated with the program inputs and return values. The first argument is an array containing a flattened parse tree. The second argument is the size of the array.

After all of the programs in a *Generation* have been assigned a cumulative fitness value, the programs are sorted in order of decreasing fitness, using a heap sort (Lewis, 1982). If this is the first or second generation of the experiment a decimation operation is optionally performed to eliminate the poorest performers. Next, a selection probability is assigned to each program. This probability is computed based on the cumulative fitness of the program relative to the cumulative fitness of the generation. The greedy over-selection technique, described earlier, is used to adjust these probabilities.

A cumulative selection probability is stored with each Program, which is the sum of that program's selection probability and the selection probabilities of all programs which appear before it in the sorted list. This cumulative probability will be used to select programs for breeding when the next generation of programs is created (as described above).

Finally, the *Generation* optionally evaluates the fitness of the "best of generation" program against the subset of tests which were not used in the earlier evaluation step. During this evaluation, the calls to evalProgram( ) are passed a *ConfusionMatrix*, which the function populates with the results of these evaluations. The *ConfusionMatrix* contains a two dimensional array of integers, where each row represents a correct result and each column represents an observed result. The array element located at row *R* and column *C* is used to record the number of times the result *C* was observed when in fact the correct result was *R*. This matrix is printed as part of the report which summarizes the performance of this generation of programs.

Also included in the report are elapsed and CPU time statistics for various operations performed by the *Generation*. These operations include the creation of the programs, their evaluation, the sorting of the programs by fitness, the decimation and the assignment of selection probabilities, and the evaluation of the "best of generation" program against the test data which was not used earlier. Elapsed time is collected using the *Timer* class, which is implemented using the Unix function gettimeofday( ). CPU time is collected using the *CPUtimer* class which uses the function clock( ).

A generation report begins with the distribution of program sizes as measured by the minimum and maximum depths of any branch within the program (this information is most useful for verifying the correctness of the ramped half-and-half algorithm in the generation of programs for the first generation). The distribution of program sizes is displayed in a table like the following, which was extracted from the results of one of the experiments performed by the author.

```
Generation 1:
   generating population

   distribution of program sizes
              Max Depth
                1    2    3    4    5    6    7    8    9   10   11+
   Min Depth
           1    0    0    0    0    0    0    0    0    0    0    0
           2    -  200   47   29   27   13    0    0    0    0    0
           3    -    -  153   28   14   13    0    0    0    0    0
           4    -    -    -  143   21   17    0    0    0    0    0
           5    -    -    -    -  138   16    0    0    0    0    0
           6    -    -    -    -    -  141    0    0    0    0    0
           7    -    -    -    -    -    -    0    0    0    0    0
           8    -    -    -    -    -    -    -    0    0    0    0
           9    -    -    -    -    -    -    -    -    0    0    0
          10    -    -    -    -    -    -    -    -    -    0    0
          11+   -    -    -    -    -    -    -    -    -    -    0
```

During the evaluation of the programs and their ranking relative to their fitness, the Generation prints status information, similar to the following:

```
evaluating programs (554 training samples)
adjusting final fitness values
sorting programs
decimating population and assigning probabilities
```

Next, the generation reports the timing data collected during the processing of this generation. These statistics are broken down by the tasks outlined earlier:

```
timing information (in seconds):

   cpu time:       generation      =     0.510
                   evaluation      =     7.280
                   sorting         =     0.020
                   assign probs    =     0.000
                   test b.o.g.     =     0.010
                   total           =     7.820

   elapsed time:   generation      =     0.507
                   evaluation      =     7.412
                   sorting         =     0.020
                   assign probs    =     0.004
                   test b.o.g.     =     0.008
                   total           =     7.951
```

The report created for a particular generation concludes with information about the top performers of the generation, including the confusion matrix which resulted from evaluation of the "best of generation" program against tests which were not used to determine selection probabilities:

```
Top 5 Programs
-------------------------------------------------------------------------
   Prog#    #Hits    Fitness    #Nodes    MaxDepth
     1        298     298.00       15          5
     2        286     286.00       17          4
     3        278     278.00       43          6
     4        277     277.00        3          2
     5        277     277.00        3          2

Best-Of-Generation Test Runs:  runs=370, hits=177, fitness=177.00

Best-Of-Generation Confusion Matrix:

            1      2      3      4     amb     (responses)
    1:     55      0     37      0      0
    2:     19     19     42     12      0
    3:     26      0     67      0      0
    4:     15     19     23     36      0
   Correct Answers:       177 ( 47.8% )
   Incorrect Answers:     193 ( 52.2% )
   Ambiguous Answers:       0 (  0.0% )
```

The last column of the confusion matrix is reserved for ambiguous answers, which were not possible in the author's chest x-ray orientation experiments. Correct answers are represented by the counts along the diagonal of the matrix, beginning at the upper left corner. In this example, the program produced a correct response 47.8% of the time, which while not particularly good, is considerably better than the 1-in-4 chance of a correct random guess.

In addition to the human readable report described above, the *Generation* also stores information about itself to a file in binary form. This information includes the statistics included in the report, as well as the parse trees for each

of the programs (to save disk space, the experimenter can request that only a subset of the programs be saved from each generation). Each parse tree is again flattened into a simple array of bytes, one byte per node. A table is written to the file which maps each byte code to a string-based name and a number of arguments. This information is used by another program, named GenerationReader, to reconstruct the parse trees.

Two other classes complete the Genetic Programming framework implemented by the author. The class *Config* parses an input file containing various parameters which control the execution of the experiment. These include the tunable parameters listed in Table 2, as well as seed values for the various random number streams needed during the experiment. *Config* provides static member functions for retrieving these values at run time.

Just as the class *Generation* manages a collection of programs, the class *Experiment<T>* manages a series of generations, thereby conducting a complete Genetic Programming experiment. The constructor for *Experiment* creates the first generation, evaluates that generation (via the eval( ) method of class *Generation*), and then enters a loop, creating and evaluating subsequent generations. The *Experiment* class collects cumulative timing statistics, which it reports following the report printed by the last *Generation*. It also displays the overall most fit program in a human readable form (by calling a method on class *Program*).

The "best of run" program is displayed as a LISP s-expression. In the example which follows (from an actual run by the author), the experiment includes five types of non-terminals: integer add ("+"), bitwise *and* ("&"), *or* ("|"), and *not* ("~"), the greater-than comparison (">"), and if-operator ("?") described earlier. There were 20 inputs named "i1" through "i20" and four constants: 0, 1, 2, and 3.

Figure 8: "Best Of Run" Program Produced During A Genetic Programming Experiment

```
(? (> i2
      i8)
   (? i2
      i16
      i11)
   (| (> (? i12
            i9
            (> i2
               (+ 1
                  i9)))
         (? (+ i17
               i4)
            i5
            i9))
      (? i15
         (+ (+ (| (| (~ 3)
                     (| (> i12
                           (? (~ i4)
                              i18
                              i18))
                        (? i10
                           i7
                           (> i8
                              i10))))
                  (? (| i11
                        (~ i13))
                     i5
                     i16))
               (~ 3))
            (~ (> (? (? (+ (| (+ i17
                                 i9)
                              (? i12
                                 i9
                                 (~ (> i7
                                       i5))))
                           i4)
                        i2
                        (| i18
                           i9))
                     i18
                     (+ 1
                        i17))
                  (| i18
                     i9))))
         (~ i2))))
```

The experimenter who uses the author's Genetic Programming framework tailors the structure of the experiment by linking the framework with modules containing the different non-terminal nodes to be included in the generated programs, by specifying tunable parameters via the configuration file, and by writing the six global functions which are used by the class *Generation* to determine the fitness of individual programs.

The data type of program inputs and outputs is defined by the exact instantiation of the *Experiment<T>* template created in the experimenter's main( ) function, which is of the form:

```
int main (int argc, char *argv[])
{
   Experiment<int> e(argc, argv);

   return 0;
}
```

In this example, the data type for the program inputs and outputs is integer and all concrete derivatives of the *Node* template which are linked with the program should be derived from the *Node<int>* instantiation.

2.2. Artificial Neural Network

The term "artificial neural network" is used to describe a variety of problem solving techniques which utilize a directed graph of relatively simple processing units. This section describes a popular class of artificial neural network, which the author used in his chest x-ray orientation experiments. The type of network is referred to as a *feedforward* network trained by *back propagation.*

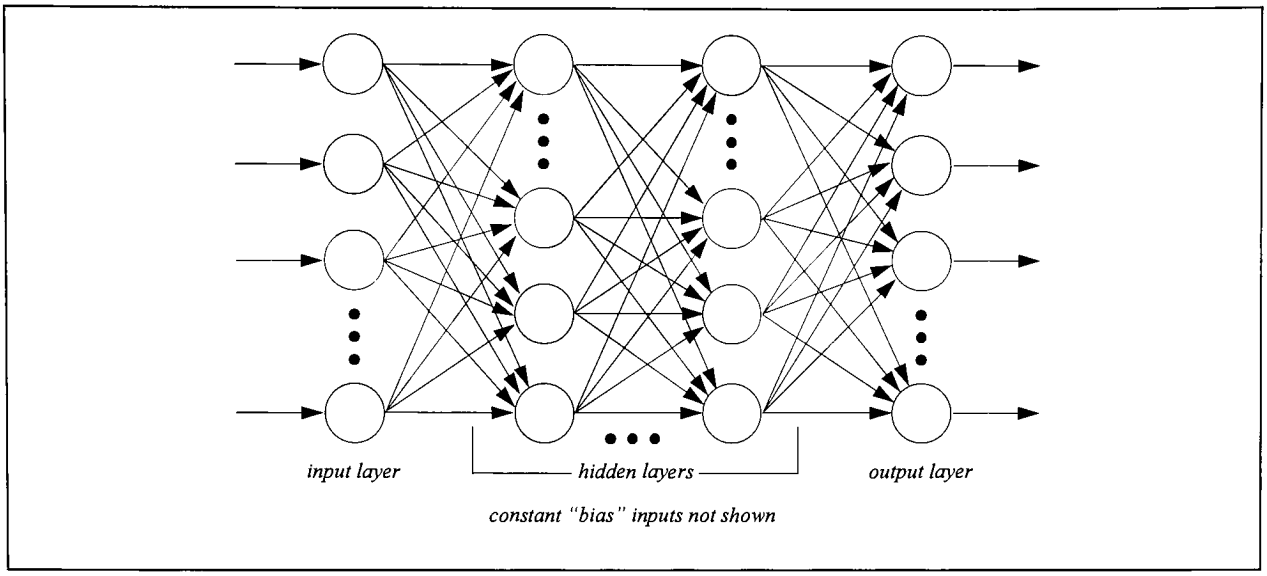
2.2.1. The Feedforward Neural Network

The general structure of a feedforward network is a directed graph of processing units grouped into an ordered collection of layers. Each processing unit in layer *N* obtains its input from the outputs of one or more processing units of layer *(N-1)* and provides its single output as an input to one or more processing units in layer *(N+1)*.

The first layer is known as the *input layer*. The processing units in the input layer obtain their inputs from a source external to the network. Normally the output of a processing unit in the input layer is exactly equal to the input supplied to it. Thus, the input layer serves simply as the place where inputs are stored while they are used by the next layer in the network (Schalkoff, 1992:237).

Feedforward networks are often applied to the problem of assigning a sample from a population to one of a fixed number of classes. The sample is represented by a vector of *features* (numeric values representing different measurements of the sample). The value of each feature is directed to a different processing unit in the input layer.

Figure 9: General Structure Of A Fully Connected Feedforward Network



The last layer of the feedforward network is known as the *output layer*. Often there is one processing unit in the output layer for each of the different classes to which a sample may be assigned. The goal is to produce a network such that the output or response produced by the processing unit associated with the true class of a sample is large while the responses produced by the other processing units in the output layer are small.

The layers of processing units which lie between the input and output layers are referred to as *hidden layers*. Feedforward networks are, in practice, *fully connected* (Bailey, 1990:43). That is, each processing unit in any layer after the first is provided a set of inputs which is exactly equal to the outputs of all of the processing units in the previous layer. In addition, a constant "bias" input is often provided to each processing unit, which allows the outputs of different processing units to differ when all of the external inputs to the network are at or near zero (Rumelhart, 1994:87). Schalkoff suggests that, internally, the processing unit can treat this bias input exactly as any other input, in which case the tuning of the bias values will occur during the adjustment of network weights, as the network is adapted to the problem being solved (Schalkoff, 1992:249).

When a processing unit in the input layer is asked for its output, it simply returns the value which has been provided to it. When a processing unit in a different layer is asked for its output, it obtains all of its inputs and from them it computes the value of its output by evaluating its *activation function*. Typically, the same form of activation function is used by all of the processing units in layers outside of the input layer.

Although a large number of suitable activation functions can be used in feedforward neural network experiments, a multi-layer network will have no advantage over a single layer network, if the activation function is linear (McClelland, 1989:131). In addition, the method of training a feedforward neural network (which will be described below) requires that the activation function have a continuous first derivative. One activation function which satisfies these criteria and is often used in feedforward neural network experiments is the *sigmoid* activation function:

Equation 1: The Sigmoid Activation Function

$$o_i = \frac{1}{1+e^{-\lambda net_i}} \qquad \text{(Equation 1.1)}$$

$$net_i = \sum_j (w_{ij}\ inp_j) \qquad \text{(Equation 1.2)}$$

where o_i is the result of the evaluation of the sigmoid function and is the output of unit i,
inp_j is the j^{th} input of processing unit i,
and w_{ij} is a constant "weight" applied to the j^{th} input of processing unit i

The constant, λ in the above equation is known as the *gain* parameter and controls the slope of the sigmoid characteristic. It is usually set to one (Schalkoff, 1992:214), as was true in all of the author's experiments. Typically the weight constants, referred to simply as the network's *weights*, are initialized to random values which are then refined through an iterative process known as *training*.

2.2.2. Training By Back Propagation

During training, the network is exposed to a number of *labeled training exemplars* (i.e., samples whose classes are known a priori). In each case, the output of the network is compared with the target output for that sample and all of the weights in the network are adjusted so that the difference between the target and observed outputs is reduced.

The adjustment of weights proceeds from the output layer toward the input layer and is known as *back propagation*. The goal of training by back propagation is to find the set of network weights minimizing the network's error function E, which describes the errors in output produced by the network as a function of the network weights.

During each training iteration, a particular weight w_{ij} is adjusted by an amount proportional to the gradient of E. The formula which determines the amount by which each weight is adjusted is known as the *generalized delta rule*.

In practice a small fraction of the adjustment from the previous training iteration is also added to the adjustment for the current iteration, to prevent oscillations in the weight adjustment strategy. This additional adjustment is known as a *momentum factor* (McClelland, 1989:136) and with the generalized delta rule leads to the following expression for the adjustment strategy:

Equation 2: The Generalized Delta Rule With Momentum

$$\Delta w_{ij}(n+1) = -\varepsilon \left(\frac{\partial E}{\partial w_{ij}} \right) + m\, \Delta w_{ij}(n) \qquad \text{(Equation 2.1)}$$

where w_{ij} is the weight being adjusted (the weight for the j^{th} input of unit i)
$\Delta w_{ij}(n+1)$ is the amount by which weight w_{ij} will be adjusted this iteration
$\Delta w_{ij}(n)$ is the amount by which weight w_{ij} was adjusted in the previous iteration
ε is a constant known as the *learning rate*
m is the momentum factor constant

Because the exact error function is unknown, a method is needed for approximating the gradient which appears in the previous equation. Schalkoff shows that the gradient can be decomposed as follows (Schalkoff, 1992:245):

Equation 3: Decomposition of Gradient Used In Generalized Delta Rule

$$\frac{\partial E}{\partial w_{ij}} = \delta_i \; inp_j \qquad \text{(Equation 3.1)}$$

where

$$\delta_i = \left(\frac{\partial E}{\partial o_i} \right) \left(\frac{\partial o_i}{\partial net_i} \right) \qquad \text{(Equation 3.2)}$$

The first factor in Equation 3.2 is equal to the difference between the observed and target outputs for unit i. The second factor is equal to the value of the derivative of the activation function evaluated at net_i. Because the target output is known for each of the processing units in the output layer, the value of δ_i for these units can be computed exactly (Schalkoff, 1992:246).

Equation 4: Derivation of δ_i For Output Layer Processing Units (sigmoid activation)

$$\frac{\partial E}{\partial o_i} = (o_i - t_i) = -(t_i - o_i) \qquad \text{(Equation 4.1)}$$

$$\frac{\partial o_i}{\partial net_i} = \lambda\, o_i\, (1 - o_i) \qquad \text{(Equation 4.2)}$$

$$\delta_i = -\lambda\, o_i\, (1 - o_i)\, (t_i - o_i) \qquad \text{(Equation 4.3)}$$

where t_i is the target output of processing unit i

Unfortunately, the target outputs of processing units in the hidden layers are not known, so the value δ_i for these units cannot be computed using Equation 4.3. Instead, a reasonable value of δ_i for a processing unit in layer N can be computed recursively from the values for δ_i computed for the units in layer (N+1), as follows (Schalkoff, 1992:247):

Equation 5: Value δ_i For Processing Units In Hidden Layers

$$\delta_i = \lambda\, o_i\, (1 - o_i) \sum_k (\delta_k\, w_{ki}) \qquad \text{(Equation 5.1)}$$

where the sum is over all processing units which accept the output of unit i as an input and w_{ki} is the weight that unit k applies to the input corresponding to i

Of note is the fact that the values of δ_i depend on the current set of weights in the next layer. Thus, a training iteration involves three passes through the network:

1. The features for the training sample are applied as inputs to the network and are fed forward through the network to compute the output of each processing unit

2. The δ_i values are computed for processing units at the output layer using Equation 4.3 and then Equation 5.1 is used to compute this value for processing units in earlier layers.

3. The values for $\Delta w_{ij}(n+1)$ are computed and added to each weight w_{ji}, to produce the new set of weights for the network

Typically, the training data must be applied to the network a number of times in order to produce an acceptable set of weights. A single application of all of the training samples is referred to as a training *epoch.* Adjustments to the weights can be done after each sample is evaluated, or the delta values computed for the weight changes can be cached and applied only at the end of an epoch (Schalkoff, 1992:241).

When weights are adjusted after each training sample (as was done in the experiments conducted by the author of this paper), it is important to randomize the order of the samples after each epoch. This eliminates the biases which can occur in weight adjustments due to similarities or differences amongst neighboring training samples. This shuffling of the training data is particularly important when momentum is included in the adjustment, since the current adjustment now depends directly on the direction and magnitude of the previous adjustment.

2.2.3. Characteristics Of A Neural Network Experiment

While the number of tunable parameters associated with an experiment involving a feedforward neural network trained with back propagation is considerably less than the number of parameters associated with a Genetic Programming experiment, it is still quite difficult to relate the values of these parameters to the characteristics of the problem being studied.

Table 3: Tunable Parameters For Artificial Neural Network Experiments

Network Topology (number of layers, processing units per layer, connectivity, etc.) Gain Parameter (λ) [assuming sigmoid activation function] Learning Rate (ε) Momentum Factor (m) Weight Adjustment Frequency (by sample vs. by epoch)

Kolmogorov's Mapping Neural Network Existence Theorem shows that any problem which can be solved by a feedforward neural network trained with back propagation can be solved by a network which contains exactly one hidden layer, which consists of a number of processing units one greater than the number of units in the input layer (Schalkoff, 1992:238). Unfortunately, the activation functions of the processing units are dependent on the nature of the problem and Kolmogorov's proof does not provide a method to construct these activation functions.

Despite these limitations, in practice, most feedforward neural networks are constructed (at least initially) with a single hidden layer (Bailey, 1990:44). The numbers of processing units in the input and output layers are determined by the number of features per sample and the number of different classes to which the samples might be assigned. As mentioned earlier, the network is typically fully connected, since it is easier to let the adjustments of weights determine the degree to which a processing unit depends on a particular input than to predict those dependencies a priori.

One might think that the number of processing units allocated to the hidden layer would be selected to agree with that described by Kolmogorov, making the design of the network topology entirely deterministic for a given experiment. In fact, this is not the case. The number of processing units in the hidden layer can affect the rate at which training converges on a good set of weights, and it can affect the degree to which the network can generalize (Bailey, 1990:46).

In the experiments which will be described in this paper, the author examines the sensitivity of this technique to three tunable parameters: number of nodes in hidden layer, learning rate, and momentum. The experiments were conducted using an object-oriented neural network simulator designed and implemented by the author in C++.

2.2.4. Implementing The Neural Network Simulator

In the simulator, the input layer is simply modeled with an array of values. Individual processing units in other layers are represented by instances of class *Node* and its derived class *oNode* (whose instances represent processing units in the output layer). The two classes differ in the way in which they compute δ_i, as one might expect from the earlier discussion.

Each instance of *Node* maintains the following data about itself:

- A pointer to the *Layer* to which it belongs (the class *Layer* will be discussed below).
- An index which identifies this nodes position in its *Layer*
- The number of inputs (including the bias input) to this processing unit
- A pointer to an array of inputs (which this *Node* shares with the other *Nodes* in its *Layer*)

- An array of weights (one weight per input, including the bias input)
- An array for holding a snapshot of weights
- The last δ_i value computed for this processing unit
- An array which contains the last values computed for $\Delta w_{ij}(n)$
- A pointer to a location where the Node's current output value can be stored

All of these data members are initialized at construction time. The *Layer* pointer, index, number of inputs, input array pointer, and output location pointer are passed to the constructor. The other arrays are allocated and initialized inside the constructor, including the array of weights, which are initialized to non-zero random numbers in the range -0.5 to 0.5 (a static member function allows a random number seed to be set for use with the system's erand48( ) function). The class oNode includes an additional data member which points to a location where the target output of the node is stored during training iterations.

Node's fire( ) method implements the sigmoid activation function with a gain parameter of one. The function assumes that the processing units in the previous layer have already been fired and that the current values in the receiver's input array are valid.

Two other member functions implement training via the generalized delta rule with momentum. The function computeDelta( ), which is implemented differently by *Node* and *oNode*, computes the value of δ_i. This function assuming that fire( ) has already been called against the nodes in the network. The other member function, adjustWeights( ), updates the weights according to equation 2.1.

Node defines four functions which are used in generating reports during the execution of an experiment. These functions print the last set of inputs to the node, the current values of the node's weights, the last snapshot of weight values, and a description of the node itself. The node description includes the current input, weight, and output values.

With the exception of the input layer, for which there are no explicit processing units modeled, the simulator stores the processing units in collections of type *Layer*. A *Layer* stores an array of pointers to its nodes and an array into

which the nodes write their outputs. A pointer to this output array is given to each node in the next layer, as that node's *input* array, thus forming the output to input connections indicated by arrows in Figure 9.

To allow for a bias input in the next layer, each instance of *Layer* allocates one additional element in the output array to which none of that layer's nodes writes a value. This element is initialized to a constant value of one. The nodes in the next layer treat this input like any other input, and the weight assigned to this input determines the "resting state" of the node (i.e., the value of the activation function when all of the normal inputs to the processing unit are at or very near zero).

The class *OutputLayer* is derived from *Layer* and differs in that it creates instances of class *oNode* instead of the base class *Node*. The output layer class also allocates an additional array of doubles, which are used during training to hold the target output values.

The class *InputLayer*, which is derived from *Layer*, also allocates two arrays of floating point values, one to hold output values (just like any other layer) and a second array which holds the input values for the network. Because an instance of *InputLayer* can contain nodes, it really models both the input and first hidden layers of the network. The number of *Nodes* constructed by an instance of *InputLayer* is independent of the number of elements allocated in the input array.

Layer objects store pointers to the next and previous layers in the network and can return pointers to the nodes they contain, given a numeric index. The class *InputLayer* implements a fire( ) method which calls fire( ) against each node in the input layer and any layers which follow. The class *OutputLayer* implements an adjustWeights( ) method which calls computeDelta( ) against each node in the network (working backwards) and then calls adjustWeights( ) against each node.

Hidden layers after the first can be modeled by inserting instances of class *Layer* between an *InputLayer* and an *OutputLayer*. In order to allow for a network which contains zero hidden layers, the simulator includes the class *InputOutputLayer*, which is derived from both *InputLayer* and *OutputLayer*. Inheritance from class *Layer* is virtual, so an instance of *InputOutputLayer* contains a single collection of nodes (one per network output), and represents a one-layer network.

Applications which use the author's neural network simulation framework do not create the *Layer* objects directly. Instead, the application creates an instance of class *Net*. The *Net* constructor is told the number of inputs to the network, the number of outputs, the number of layers (one or more), and the number of processing units in each of the hidden layers. The constructor creates the appropriate instances of the *Layer* classes and maintains pointers to the first and last.

The *Net* object provides pointers to three arrays, which are actually allocated by the input and output layer objects. These arrays include the inputs to the network, the latest outputs from the network, and the target outputs for the network. The class includes a flag which indicates whether training is "on" or "off". The fire( ) method of the *Net* calls fire( ) against the *InputLayer*, and if training is on, it then calls adjustWeights( ) against the *OutputLayer*. The application must populate the input array and target output array (if training is on) before calling fire( ).

Using this framework of classes the author of this paper implemented a neural network simulator for solving the chest x-ray orientation problem. The simulator reads a configuration file which determines the following characteristics of the experiment:

- the number of layers in the network
- the number of nodes in each hidden layer
- the learning rate parameter
- the momentum factor parameter
- the fraction of labeled exemplars which are used for training
- the number of training iterations to execute
- the frequency at which progress is reported to the user of the simulator
- a random number seed for the random number stream which is used to assign the initial network weights
- a random number seed for the random number stream used to shuffle the samples during training.

The labeled training samples (which are read from a separate file) are divided into two groups. The first group is used for training (i.e., adjusting the weights of the network). The second group is used to evaluate the ability of the network to generalize. The network weights are never adjusted based on these latter samples.

The program reads the feature data for all of the samples into memory. For each of the different features a minimum and maximum value is determined by looking at the collection of labeled samples which are used during training. These minimum and maximum values are then used to normalize the feature values (i.e., adjust the values so that they are always in the range 0.0 to 1.0).

After the feature data has been normalized, the program creates a *Net* object and begins training it with first group of samples. The feature values are stored in a two dimensional array, where the first index is a sample number and the second index identifies a feature for that sample. The samples are shuffled prior to each training iteration by randomizing an array of values corresponding to the first index of the feature matrix.

After a given number of training iterations (determined by data in the configuration file), the program displays a snapshot of the current state of the network in the form of two confusion matrices. The first confusion matrix displays the degree to which the network has successfully classified the training samples. The second matrix displays the performance of the network against the samples which are not used for training the network. This latter matrix indicates the degree to which the network is able to generalize.

Figure 10: Sample Output For One Snapshot Of Neural Network Simulator

```
Training Pass 50 (2 runs):
              0    1    2    3
      0 :   137    0    2    0
      1 :     0  138    0    1
      2 :     1    0  137    0
      3 :     0    0    0  138
    Right Answers:  550 ( 99.3 percent )
    Wrong Answers:    4 (  0.7 percent )
    Ambiguous:        0 (  0.0 percent )

    Elapsed Time = 29 seconds; CPU Time = 28 seconds

    Test Samples (non-training data)
              0    1    2    3
      0 :    91    0    1    0
      1 :     0   90    0    2
      2 :     1    0   92    0
      3 :     0    1    0   92
    Right Answers:  365 ( 98.6 percent )
    Wrong Answers:    5 (  1.4 percent )
    Ambiguous:        0 (  0.0 percent )

    Time To Run Test Samples = 69ms (70ms CPU)

    Best So Far Is Pass 42 With 99.1% Correct
```

A correct answer is one in which the output for the node corresponding to the true class of the sample is greater than 0.5 and the output for the other nodes in the output layer are all less than 0.5. If two or more output nodes return a value of 0.5 or greater, the result is labeled ambiguous. The code which implements the confusion matrix in this program does not collect data on the number of ambiguous responses by output class. Target outputs during training are set to 1.0 for the correct answer and 0.0 for the incorrect answers.

During the run, the neural network simulator created for the chest x-ray orientation problem collects CPU and elapsed time statistics for the period during which the network is being trained. The cumulative values for these statistics are reported during each snapshot (near the center of the sample output in Figure 10) and are used when comparing the cost of using the different pattern classification techniques presented in this paper.

2.3. Probabilistic Neural Network (PNN)

The two classification techniques described thus far used iterative approaches to improving the accuracy of the classifier. Specht describes a different kind of neural network classifier, the *probabilistic neural network* (PNN), which in theory, does not rely on iterative training, because the training data is actually stored in the network as the network is constructed (Specht, 1988:525). Chettri and Cromp, show how an iterative technique of refinement can be employed to reduce the number of exemplars which need to be stored in the network, reducing the time required to evaluate the network for unknown samples (Chettri, 1993:187).

2.3.1. The Decision Rule

The PNN algorithm relies on the maximum a posteriori decision rule. Given M classes, let $f(\boldsymbol{X}|\, S_k)$ be the probability density function associated with the measurement vector [i.e., sample] $\boldsymbol{X}$, given that $\boldsymbol{X}$ is from class k. Further, let $P(S_k)$ be the a priori probability that a random sample will belong to class S_k. We then have the following decision rule (Chettri, 1993:189):

Equation 6: Maximum A Posteriori Decision Rule (assigns X to class S_k if and only if...)

$$\delta_k(\mathrm{X}) \geq \delta_j(\mathrm{X}) \quad \text{for all } j \in \{0,1,\ldots, M-1\} \qquad \text{(Equation 6.1)}$$

where

$$\delta_k(\mathrm{X}) = f(\mathrm{X}|S_k)P(S_k) \qquad \text{(Equation 6.2)}$$

Equation 6.2 defines what is known as the *discriminant* function for class k. The description of this discriminant function by Specht includes a factor which can be used to weight the values of the δ_k relative to the differences in cost of the possible types of misclassification errors (e.g., assigning a sample to class S_j when it is really of class S_i) (Specht, 1988:526). Although it is often the case that the a priori probability $P(S_k)$ will be known for each class, it is unusual to know the probability density function $f(\boldsymbol{X}|\, S_k)$, which is also a factor in the discriminant.

A pattern classification technique which is similar to the PNN technique is the Gaussian Maximum Likelihood Classifier (GMLC) (Schalkoff, 1992:61). The GMLC assumes that the probability density function (pdf) is a normal multivariate pdf, given by

Equation 7: PDF For Gaussian Maximum Likelihood Classifier

$$f(\mathrm{X}|S_k) = \frac{\exp\left[-\frac{1}{2}(\mathrm{X}-\mu)^T \Sigma^{-1}(\mathrm{X}-\mu)\right]}{(2\pi)^{d/2}\left|\Sigma\right|^{1/2}} \quad \text{(Equation 7.1)}$$

where $\boldsymbol{X}$ and μ are each dx1 vectors and μ and Σ are the population mean and (dxd) covariance matrix for class k.

The PNN described by Specht uses an estimate for the pdf which does not rely on the values for the population mean and covariance matrix (Specht, 1988:526). This results in a significant reduction in the time required to construct the classifier, but can make the evaluation of the classifier less computationally efficient than the GMLC.

Equation 8: PDF For Probabilistic Neural Network Classifier

$$f(\mathrm{X}|S_k) = \frac{1}{(2\pi)^{d/2}\,\sigma^d}\,\frac{1}{P_k}\sum_{i=1}^{P_k}\exp\left[-\frac{(\mathrm{X}-W_{ki})^T(\mathrm{X}-W_{ki})}{2\sigma^2}\right] \quad \text{(Equation 8.1)}$$

where d is the number of features associated with a sample
P_k is the number of exemplars of class k stored in the network
W_{ki} is the i^{th} exemplar of class k
and σ is a tunable "smoothing" parameter

Specht describe a feedforward neural network architecture in which each processing unit is responsible for computing part of the discriminant function defined by Equation 6.2 and Equation 8.1 for one of the classes to which a sample might be assigned (Specht, 1988:527). Although it may make sense to implement such a network in hardware, there is no apparent reason for a software implementation to distribute the computation of a single discriminant across multiple objects.

If the a priori probability of each class is the same and the number of stored exemplars is the same for each class, then the discriminant function for class k can be simplified to the following:

Equation 9: Simplified Discriminant Function For Restricted Set Of PNN's

$$\delta_k(\mathrm{X}) = \sum_{i}^{P_k}\exp\left[-\frac{(\mathrm{X}-W_{ki})^T(\mathrm{X}-W_{ki})}{2\sigma^2}\right] \quad \text{(Equation 9.1)}$$

Equation 9 is really a measure of the distance between the sample being classified and the various exemplars stored in the network. If an exemplar is *close* to the sample, then the numerator in Equation 9 is small and corresponding component of the sum is large. As the distance between the sample and exemplar increases, the component of the sum decreases. Thus the class for which the discriminant function is largest is exactly the class whose exemplars are closest to the sample.

Even in this simplified version of the PNN discriminant function, the cost of computing the discriminant increases linearly as the number of exemplars stored in the network increases. Thus the goal in constructing a PNN is to select smallest set of exemplars from each class which are sufficient to distinguish samples of that class.

2.3.2. Learning Vector Quantization (LVQ)

To construct PNN's in which only a small number of exemplars are stored in the network, Chettri and Cromp used the learning vector quantization (LVQ) technique to iteratively refine a small set of exemplars, selected randomly from a much larger set of training data (Chettri, 1993:193). This technique does not directly affect the number of exemplars stored in the network. Instead, LVQ modifies an existing set of exemplars to improve their classification ability. As a result, a small number of exemplars refined by LVQ can be used in place of a larger number of unrefined exemplars.

The LVQ technique requires a number of passes through the full set of training data, during which each training sample is used to adjust the value of one exemplar stored in the network. Each pass through the training data proceeds as follows:

1. Randomly shuffle the training data.
2. For a training sample $\boldsymbol{X}$, locate the stored exemplar W_{ki} which is *closest* to $\boldsymbol{X}$
3. If $\boldsymbol{X}$ is of class S_k, adjust W_{ki} so that it moves closer to $\boldsymbol{X}$. Otherwise, adjust W_{ki} so that it moves further from $\boldsymbol{X}$.
4. Repeat steps 2 and 3 for the next training sample, until there are no more samples.

The measure of *closeness* is simply the distance between the two vectors $\boldsymbol{X}$ and W_{ki}. The amount by which W_{ki} is adjusted is given by:

Equation 10: Adjustment To Closest Exemplar During LVQ Training

$$\Delta W_{ki} = \begin{cases} +\alpha(t)\left[X - W_{ki}\right] & \text{if } X \in S_k \\ -\alpha(t)\left[X - W_{ki}\right] & \text{if } X \notin S_k \end{cases} \qquad \text{(Equation 10.1)}$$

where $\alpha(t)$ is the "learning rate", which is a positive, monotonically decreasing function of the training iteration number (i.e., the number of passes through the training data).

2.3.3. Characteristics Of A PNN With LVQ

When coupled with LVQ, the PNN for a particular sample space is determined by a relatively small number of tunable parameters:

Table 4: Tunable Parameters For PNN Experiments

Number Of Exemplars Stored Per Class
Smoothing Parameter (σ)
LVQ Learning Rate Function ($\alpha(t)$)
Number Of Training Iterations During LVQ

Chettri and Cromp suggest that the number of training iterations is typically between 500 and 10,000. They further suggest that a good value for $\alpha(t)$ is $1/t$. In their experiments, conducted with this learning rate, they observed little improvement after 500 training iterations (Chettri, 1993:193).

Specht claims that it is not difficult to find a reasonable value for the smoothing parameter, σ, which appears in Equation 8.1 and Equation 9.1, and that small changes to this parameter do not significantly affect the success of the classifier. In addition, as the value of this parameter approaches zero, the PNN approximates a "nearest neighbor" classifier, in which a sample is assigned to the class from which was drawn that exemplar which is closest to the sample. (Specht, 1990:113). Chettri and Cromp refer to work by Koontz and Fukunaga (Koontz, 1972) in which the value of σ was computed from the training data. As Chettri and Cromp point out, the calculation requires computing the covariance matrix, which the PNN technique was designed to avoid (Chettri 1993:191).

2.3.4. Implementing The PNN Simulator

In the PNN simulator designed by the author of this paper, each of the parameters in Table 4 is configurable, including the learning rate function, for which the program allows a certain degree of flexibility (described below).

Like the author's feedforward neural network simulator, the behavior of the PNN simulator is controlled by data stored in two configuration files. The first file contains the values of parameters which are specific to the PNN and LVQ algorithms, including:

- the number of exemplars per class which will be stored in the network
- the number of training iterations
- the percent of all labeled sample data which is actually used for LVQ
- the value of the smoothing parameter, σ
- a constant, *lr*, which is used in the second of two learning rate functions
- a flag which indicates which of the two forms of learning rate functions to use
- random number seeds for selecting initial exemplars and for shuffling training samples

The two forms of learning rate functions supported by the simulator are as follows:

Equation 11: Learning Rate Functions Supported By The PNN Simulator

$$\alpha(t) = 1 / t \qquad \text{(Equation 11.1)}$$

$$\alpha(t) = (1 + lr) / (1 + t) \qquad \text{(Equation 11.2)}$$

where *lr* is a constant, tunable learning rate parameter.

The first of these equations is identical to that used by Chettri and Cromp. During the first training pass, the value of this function is 1.0 and simulator is simply swapping the exemplars for other training samples. The author of this paper decided to effectively start training at iteration number two, and so the denominator of Equation 11.2 is simply `(1 + t)`. The learning rate parameter, *lr*, provides some control over the how quickly the learning rate function approaches zero.

The second configuration file used by the PNN simulator contains the labeled training data in the same format used by the other programs described in this paper. As with the other programs implemented by the author, these samples are divided into two groups: samples used for training of the classifier and samples used to test the ability of the classifier to generalize. The relative sizes of these two groups is determined by a parameter in the first configuration file.

In order to prevent floating point underflows during the calculation of the discriminant function, the PNN simulator normalizes each component of the training sample vectors. When the sample data is read from the second configuration file, the program keeps track of the minimum and maximum observed value for each of the different features amongst those samples which are used during training. These minimum and maximum values are then used to map each feature value into the range zero to one.

Once the data is normalized, the correct number of exemplars for each class are randomly selected from amongst the samples which are available for training. Training then proceeds as described earlier, using one of the two forms of learning rate function. The program periodically monitors the progress of the training by evaluating a PNN based on the current set of exemplars against all of the available samples. After each of these points of evaluation, the program displays a summary of the current performance in terms of a pair of confusion matrices:

Figure 11: Sample Output For One Snapshot Of PNN Simulator

```
Snapshot 3 (after 5 training iterations)

   Training Data Results:
              0     1     2     3
     0 :    134     2     1     4
     1 :      0   137     2     1
     2 :      2     0   136     2
     3 :      1     1     1   137
   Right Answers:   544 (  97.0 percent )
   Wrong Answers:    17 (   3.0 percent )

   Test Data Results:
              0     1     2     3
     0 :     83     8     1     1
     1 :      1    82    10     1
     2 :      2     0    83     9
     3 :      7     1     1    85
   Right Answers:   333 (  88.8 percent )
   Wrong Answers:    42 (  11.2 percent )

   Cum. Training Time: 5972 msecs (2290 cpu msecs)

   Cum. Evaluation Time: 6071 msecs (2380 cpu msecs)

   Best Snapshot So Far Is #2 (93.8 percent correct )
```

After the configured number of training iterations are performed, the PNN is evaluated one last time against the training and test data and a final pair of confusion matrices is printed. Each snapshot includes cumulative elapsed and CPU time statistics for training and evaluation of the PNN. At the end of the run, the current set of exemplars are written to a file. In addition, the set of exemplars which corresponded to the best snapshot are also written to the file.

2.4. Linear Pattern Classifier

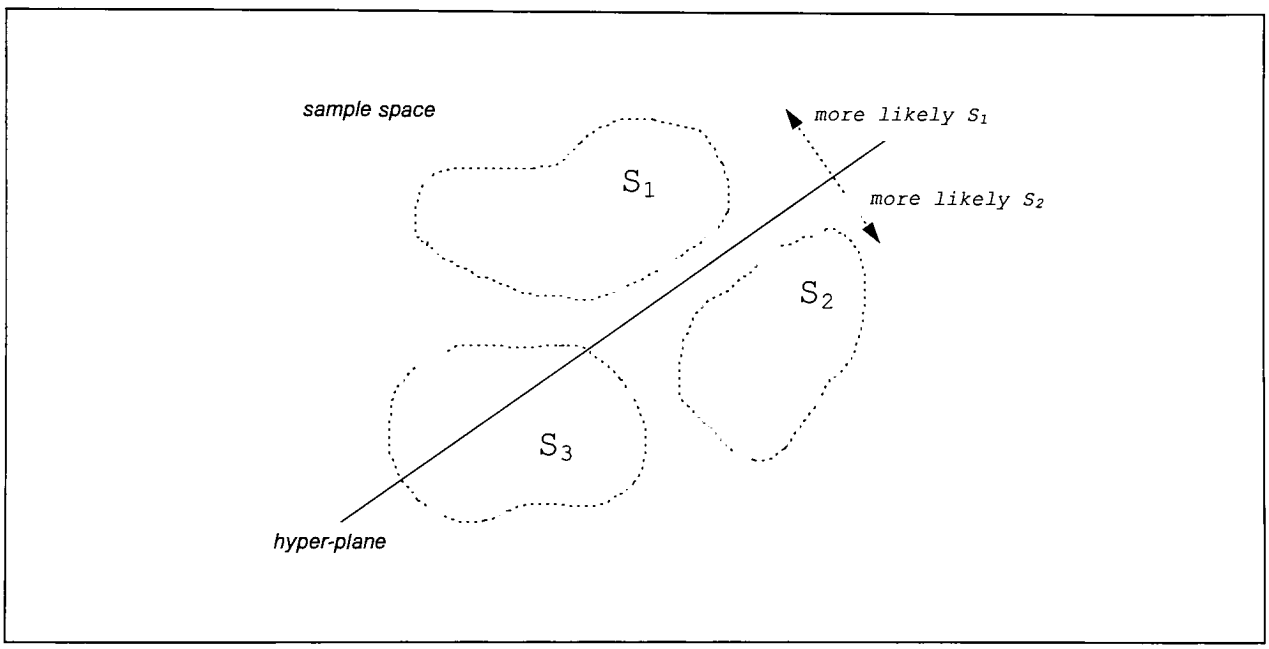
Of the four pattern classification techniques described in this paper, the *linear pattern classifier* is by far the most computationally efficient classifier to construct. Like the probabilistic neural network, the training exemplars are stored in the classifier at construction time. Unlike the other three techniques however, no additional training of the classifier is performed.

2.4.1. Decision Boundaries

Where the sample space is represented by d-dimensional feature vectors, the linear classifier works by defining a number of d-dimensional hyper-planes, each of which divides the sample space into two halves. One hyper-plane is defined for each unique pairing of possible classes to which a sample might be assigned.

The hyper-plane associated with classes S_1 and S_2 is a boundary for deciding to which of the two classes an unknown sample $\boldsymbol{X}$ is more likely to belong. All samples which lie on one side of the hyper-plane are considered more likely to be of class S_1, while the samples on the other side are considered more likely to be of class S_2.

Figure 12: 2D Example Of Hyper-Plane Decision Boundary (Classes S_1 and S_2)

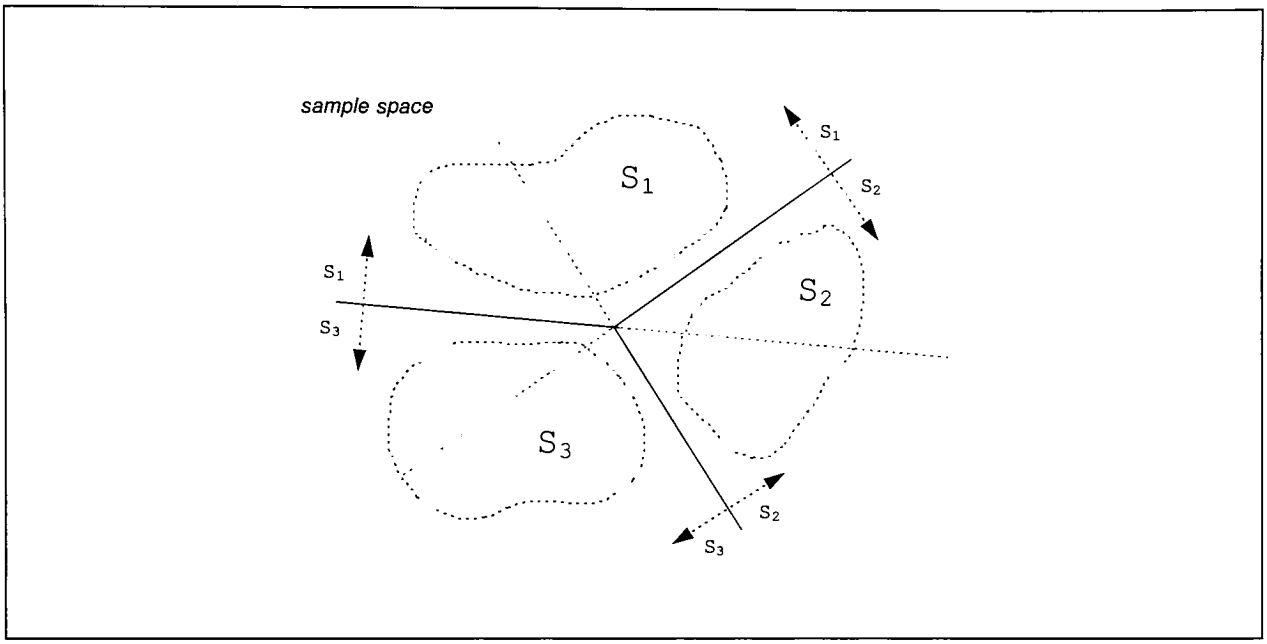


As Figure 12 illustrates, a particular hyper-plane provides no information about classes other than the two for which it serves as a decision boundary. By itself, the hyper-plane in the figure is insufficient to make a decision about the exact class of an unknown sample.

However, the collection of all hyper-planes defined by the linear classifier divides the sample space into a number of regions. In some regions, one class will be more likely than all of the others. A sample which is found to lie in such a region is assigned to this most likely class. In Figure 13, the three hyper-planes divide the sample space into six regions. The two uppermost regions form a larger region (bounded by the solid portions of the hyper-planes), in which samples are more likely to be of class S_1 than either class S_2 or S_3. Any sample found to lie in this region

would be assigned to class S_1. Even though some samples in this region are more likely of class S_2 than S_3 (and vice versa), S_1 is more likely than either of these two classes, and the entire region is assigned to S_1.

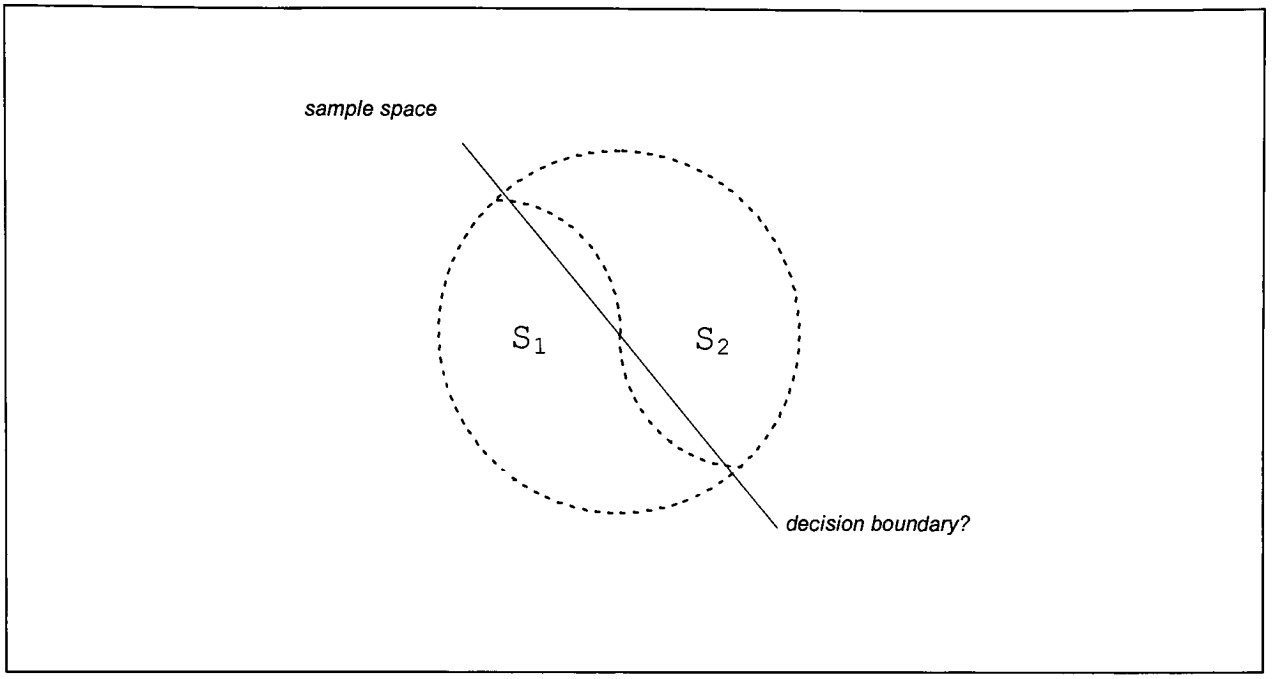
Figure 13: Sample Space Divided Into Regions By Three Hyper-Planes



It is possible that the collection of hyper-planes will result in regions where ambiguities exist as to which class is more likely than the others. In some cases, it may not even be possible to separate the different classes with hyper-plane boundaries, as in the example shown in Figure 14.

Despite the fact that the two classes in Figure 14 cannot be exactly separated by a hyper-plane decision boundary, a classifier based on the hyper-plane shown would still do a reasonable job classifying the majority of samples of either class.

Figure 14: Classes Which Cannot Be Separated By A Hyper-Plane Boundary



2.4.2. The Linear Classifier

The creation of the pair-wise class decision boundaries and their use in classifying samples is implicit in the linear classifier algorithm. The linear classifier is a `(d+1) x M` matrix, W, where `d` is the number of features in a sample and `M` is the number of classes.

To classify an unknown sample X, one converts the d-dimensional vector, X, to a (d+1)-dimensional vector, X', by appending a value of 1.0 to the end of the vector (this converts the vector to its standard "homogeneous" representation, and is necessary in order to allow decision boundaries which do not pass through the origin of the sample space). The vector X' is post-multiplied by W, resulting in a `1 x M` *result* vector, R. The result vector contains a response value for each class. The sample X is assigned to the class which exhibits the greatest response.

Given n labeled training exemplars, $\{ X_1, X_2, \ldots, X_n \}$, the linear classifier matrix, W, is constructed by solving Equation 12.1 (really a system of linear equations) (Schalkoff, 1992:99):

Equation 12: Linear Classifier Equation

$$AW = B \qquad \text{(Equation 12.1)}$$

where

$$A = \begin{bmatrix} X_1 & 1 \\ X_2 & 1 \\ \vdots & \vdots \\ X_n & 1 \end{bmatrix} = \begin{bmatrix} x_1^1 & x_1^2 & \cdots & x_1^d & 1 \\ x_2^1 & x_2^2 & \cdots & x_2^d & 1 \\ \vdots & \vdots & & \vdots & \vdots \\ x_n^1 & x_n^2 & \cdots & x_n^d & 1 \end{bmatrix} \qquad \text{(Equation 12.2)}$$

$$B = \begin{bmatrix} R_1 \\ R_2 \\ \vdots \\ R_n \end{bmatrix} = \begin{bmatrix} r_1^1 & r_1^2 & \cdots & r_1^M \\ r_2^1 & r_2^2 & \cdots & r_2^M \\ \vdots & \vdots & & \vdots \\ r_n^1 & r_n^2 & \cdots & r_n^M \end{bmatrix} \text{ and } r_i^k = \begin{cases} +1 \text{ if } X_i \in S_k \\ -1 \text{ if } X_i \notin S_k \end{cases} \qquad \text{(Equation 12.3)}$$

The matrix $\mathbf{A}$ contains all of the training samples, converted to homogeneous coordinates. Each row of the matrix $\mathbf{B}$ is the target response vector for the training sample in the corresponding row of $\mathbf{A}$. A row in $\mathbf{B}$ contains -1 in each column which corresponds to a class other than the true class of the sample. The column of $\mathbf{B}$ which corresponds to the true class of the training sample contains +1. The choice of values, +1 and -1, is not critical, provided that the value for the correct class is larger.

Equation 12.1 can be solved for $\mathbf{W}$ by pre-multiplying both sides of the equation by the inverse of $\mathbf{A}$. Since $\mathbf{A}$ is not usually square, one must resort to using its *pseudo-inverse*, $\mathbf{A}^\dagger$, which is a matrix that minimizes $\| (\mathbf{A}^\dagger)\mathbf{A} - \mathbf{I} \|^2$ (Schalkoff, 1992:291).

Equation 13: Pseudo-Inverse Solution For Finding Linear Classifier Matrix $\mathbf{W}$

$$W = A^\dagger B = \left[\left(A^T A \right)^{-1} A^T \right] B \qquad \text{(Equation 13.1)}$$

It is possible that the pseudo-inverse of the matrix $\mathbf{A}$ cannot be computed, because the product of $\mathbf{A}$ and its transpose (see Equation 13.1) may be singular. In this case, the linear classifier cannot be constructed. A common cause for this failure is duplicate training samples in $\mathbf{A}$. During the author's chest x-ray orientation experiments, duplicate

samples were found. When these duplicates were removed, the linear classification matrix was successfully computed.

Each column w_i of W defines a hyper-plane decision boundary which, by itself, divides the sample space into two regions: those samples which are likely to be of class S_i and those which are not.

Equation 14: Decision Boundary From One Column Of Linear Classifier Matrix W

$$X^{'} w_i - b = \begin{cases} > 0 \text{ if } X \in S_i \text{ is likely} \\ < 0 \text{ if } X \notin S_i \text{ is likely} \end{cases} \qquad \text{(Equation 14.1)}$$

where b is one-half the sum of the two different values used as elements in the matrix B

Thus each response value in the result vector, R, is a measure of the likelihood that the sample X belongs to one of the classes. When two of these values are compared, the greater of the two values indicates which of the two classes is more likely to be the true class of sample X.

Equation 15: Comparing Linear Classifier Result Values For Two Classes

$$X^{'} w_i - X^{'} w_j = X^{'}(w_i - w_j) = \begin{cases} > 0 \text{ if } X \in S_i \text{ more likely} \\ < 0 \text{ if } X \in S_j \text{ more likely} \end{cases} \qquad \text{(Equation 15.1)}$$

Equation 15.1 corresponds to the hyper-plane decision boundaries illustrated in Figure 12 and Figure 13, at the beginning of this section. When all of the values in the result vector are compared and the class is assigned based on the largest of these values, the classifier is making a decision based on the location of the sample, relative to each of these pair-wise class decision boundaries.

2.4.3. Implementing The Linear Classifier

In the author's implementation of the linear classifier algorithm, a *Matrix* class was unplemented to hold the training data, test data, linear classifier, and classification results. The number of rows and columns in the matrix is defined at construction time. *Matrix* includes methods to:

- initialize a square matrix to the identity matrix
- get or set a particular element

- copy a specified matrix (assignment operator and copy constructor)
- post-multiply a matrix by a specified matrix
- add two matrices together
- produce the transpose of a matrix
- produce the inverse or pseudo-inverse of a matrix, if it exists

The matrix data is stored separate from the *Matrix* object and is reference counted, so that it can be shared by multiple instances of class *Matrix*. Operations like setElement( ) will first copy the matrix data if the reference count is greater than one. All of the operations above are fairly straightforward, with the exception of the calculation of the inverse or pseudo-inverse.

The book, *Numerical Recipes In C*, describes a method for computing matrix products of the form $(\boldsymbol{P}^{-1}\,\boldsymbol{Q})$ given the *LU-decomposition* of a rowwise permutation of the matrix $\boldsymbol{P}$ (Press, 1992:48). If $\boldsymbol{Q}$ is the identity matrix, the result of this calculation is the inverse of $\boldsymbol{P}$. If $\boldsymbol{P}$ is equal to $(\boldsymbol{A}^T\boldsymbol{A})$ and $\boldsymbol{Q}$ is equal to $\boldsymbol{A}^T$, then the result is the pseudo-inverse $\boldsymbol{A}^\dagger$ (see Equation 13.1). The book also describes *Crout's algorithm* for producing the necessary LU-decomposition (Press, 1992:43).

The LU-decomposition of a matrix, $\boldsymbol{P}$, is a pair of matrices, $\boldsymbol{L}$ and $\boldsymbol{U}$, whose product is $\boldsymbol{P}$ and where $\boldsymbol{L}$ is lower-triangular (all elements above the diagonal are zero) and $\boldsymbol{U}$ is upper-triangular (all elements below the diagonal are zero). Equation 16 illustrates the LU-decomposition of a matrix.

Equation 16: Example Of LU-Decomposition Of A 4x4 Matrix

$$L\ U\ =\ P\ =\ \begin{bmatrix} p_{11} & p_{12} & p_{13} & p_{14} \\ p_{21} & p_{22} & p_{23} & p_{24} \\ p_{31} & p_{32} & p_{33} & p_{34} \\ p_{41} & p_{42} & p_{43} & p_{44} \end{bmatrix} \qquad \text{(Equation 16.1)}$$

where

$$L = \begin{bmatrix} \alpha_{11} & 0 & 0 & 0 \\ \alpha_{21} & \alpha_{22} & 0 & 0 \\ \alpha_{31} & \alpha_{32} & \alpha_{33} & 0 \\ \alpha_{41} & \alpha_{42} & \alpha_{43} & \alpha_{44} \end{bmatrix} \qquad U = \begin{bmatrix} \beta_{11} & \beta_{12} & \beta_{13} & \beta_{14} \\ 0 & \beta_{22} & \beta_{23} & \beta_{24} \\ 0 & 0 & \beta_{33} & \beta_{34} \\ 0 & 0 & 0 & \beta_{44} \end{bmatrix} \qquad \text{(Equation 16.2)}$$

Crout's algorithm actually produces the LU-decomposition for a rowwise permutation of the input matrix, P. This is sufficient for computing the target product $(P^{-1} Q)$, provided that information about the re-ordering of rows is maintained along with L and U.

The author's linear classifier program reads the labeled training exemplars from a file and populates the matrices A and B from Equation 12.2 and Equation 12.3. A command line argument determines the percentage of training samples used to compute the linear classification matrix W. The remaining samples are used to evaluate the ability of the classifier to generalize.

In addition to writing the classification matrix to a file, the linear classifier program also displays a pair of confusion matrices, which summarize the classifier's performance against the samples used to construct it and the samples which were later used to test it. An ambiguous answer is one where the strongest response exhibited by the classifier was shared by two or more classes. Following the confusion matrices, the program displays the elapsed and CPU time required to construct the matrix and to classify all of the sample data.

Figure 15: Example Of Output From Author's Linear Classifier Program

```
Confusion Matrix For Training Samples:

          1      2      3      4      ?      (responses)
1:      184      0      1      0      0
2:        0    184      0      1      0
3:        1      0    184      0      0
4:        0      2      0    182      0
Correct Answers:      731 ( 99.3% )
Incorrect Answers:      5 (  0.7% )
Ambiguous Answers:      0 (  0.0% )

Confusion Matrix For Test Samples:

          1      2      3      4      ?      (responses)
1:       46      0      0      0      0
2:        0     46      0      0      0
3:        0      0     46      0      0
4:        0      0      0     46      1
Correct Answers:      184 ( 99.5% )
Incorrect Answers:      0 (  0.0% )
Ambiguous Answers:      1 (  0.5% )

Matrix Generation:    569ms (   470ms CPU time)
Matrix Evaluation:    157ms (    60ms CPU time)
```

3. Chest X-Ray Feature Extraction:

Each of the four classification techniques which have been described in this paper assigns a sample from some population to one of a number of classes into which the population can be segregated. The sample is presented to the classifier in the form of a vector of *feature* values. Each position in the feature vector represents some measurement which can be made against the sample, and each feature value is the result of the corresponding measurement against the sample being classified.

The choice of features which will be used to represent a sample is critical to the success of the classifier. In designing a feature set, the experimenter strives to select features which will distinguish samples from different classes, while minimizing the apparent differences between samples of the same class. Often there is a great deal of uncertainty regarding which features are important to the classification problem. Selecting the appropriate feature set is often the most difficult step in the development of a classification system (Rumelhart, 1994:91).

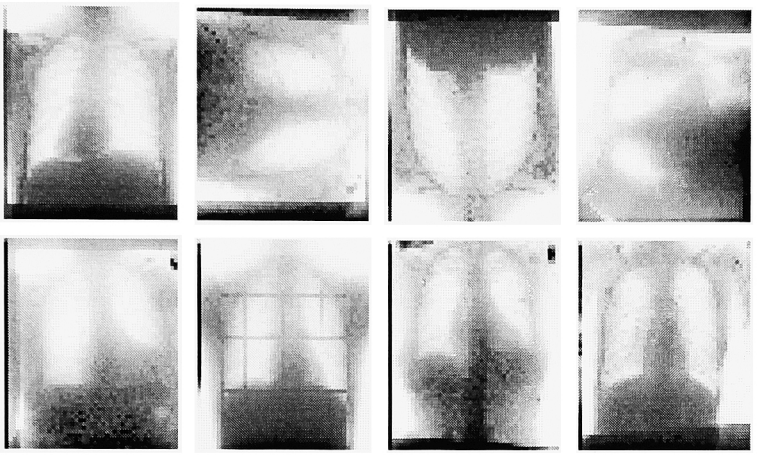
The sample space for the author's experiments consists of gray-scale images of chest x-rays, which are either oriented correctly (i.e., right-side-up) or are rotated through some integral multiple of 90 degrees. The classifiers are designed to assign a sample image to one of four classes, where each class represents one of the four possible orientations.

When the x-ray is taken, x-rays pass through the patient and strike a rectangular plate which is coated with an x-ray sensitive material that is scanned by specialized hardware and converted into a gray-scale digital image. Because the plate is rectangular, the technician may intentionally orient the plate incorrectly, in order to capture more of the patient's chest. In other cases, the plate may be inadvertently oriented incorrectly.

Figure 16 provides some examples of chest x-rays from the images used in the author's experiments. To enhance readability of Figure 16, the images have been scaled and gamma corrected (a modification of the brightness curve of the image) using *Microsoft Imager*.

Each sample image is actually a two-dimensional grid of picture elements (i.e., *pixels*), each of which is characterized by an intensity value in the range zero (black) through 255 (white). Each image is 45 pixels wide and 55 pixels high. The images were produced by sub-sampling larger images which were captured at the time the x-rays were taken.

Figure 16: Examples Of Chest X-Ray Images (enhanced)



One possible set of features which could be used to describe a sample image are the intensity values of the 2,475 pixels which comprise the image. This would provide the classifier with all available information for making a decision as to the proper orientation for the x-ray. However, this is a rather large amount of raw data, and it is unrealistic to assume that most classification techniques would yield a good classifier in a reasonable amount of time, when faced with this much data.

For example, in a genetic programming experiment where the average number of arguments to each operator is two, a single parse tree would need to contain a minimum of (almost) 5,000 nodes, in order to reference each of the feature values just once. The 85-node parse tree from Figure 8, on the other hand, references many of its input values multiple times. This parse tree was the result of an experiment in which there were only 24 feature values per sample. That experiment required almost 2,000 seconds of CPU time to execute.

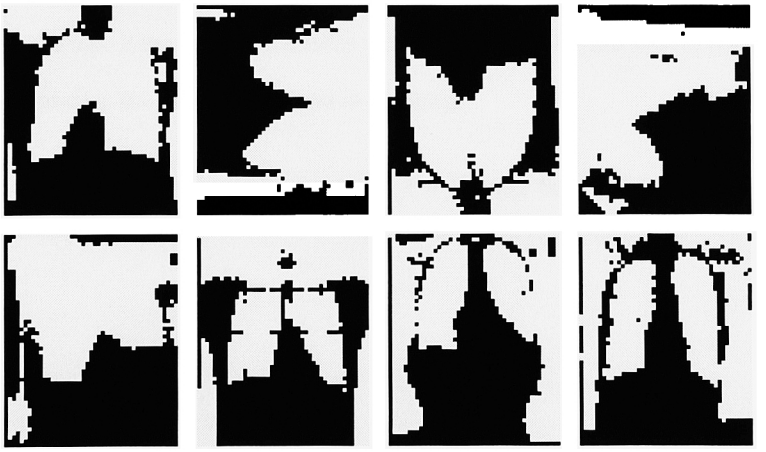
Instead of using the raw pixel data as the feature set, the author implemented a pair of feature extraction programs, each of which reads the sample images into memory and generates a set of feature vectors, containing 20 features per sample. The two programs differ in the way in which the 20 feature values can be related back to the original 45 x 55 pixel, x-ray image. Each of the four different classification techniques was used to construct classifiers for each of the two feature sets. The two feature sets will be referred to as F_1 and F_2.

3.1. Common Elements Of The Two Feature Sets

In both of the feature extraction programs, the sample image is first *cropped* about its center, to produce a square image, so that differences between the width and height of a sample cannot influence the classification exercise. Cropping involves discarding pixels which lie outside a pre-defined region. For feature set F_1, the sample image is cropped to `21 x 21` pixels. For feature set F_2, the cropped image size is `45 x 45` pixels.

The cropped image is converted to a *bitonal* image, by performing a *threshold* operation (Gonzalez, 1987:354). A bitonal image is one in which the intensity of each pixel is either zero or 255 (the code values for black and white, respectively). For each pixel in the source image, the threshold operation replaces the pixel's value with zero, if the intensity of the pixel is less than a constant known as the *threshold value*. If the pixel's intensity is greater than or equal to the threshold value, the operation replaces the pixel value with 255.

Figure 17: Example Of Threshold Operation Applied To X-Ray Image



By properly selecting a threshold value, it is possible to convert the sample into an image in which, at the center of the image, only the lungs are visible. Looking at the shape of the lungs and their position in the bitonal image, the author was (in general) able to determine the proper orientation of the image by visual inspection. From this, the author concluded that the bitonal image contains sufficient information with which a classifier might correctly orient the sample.

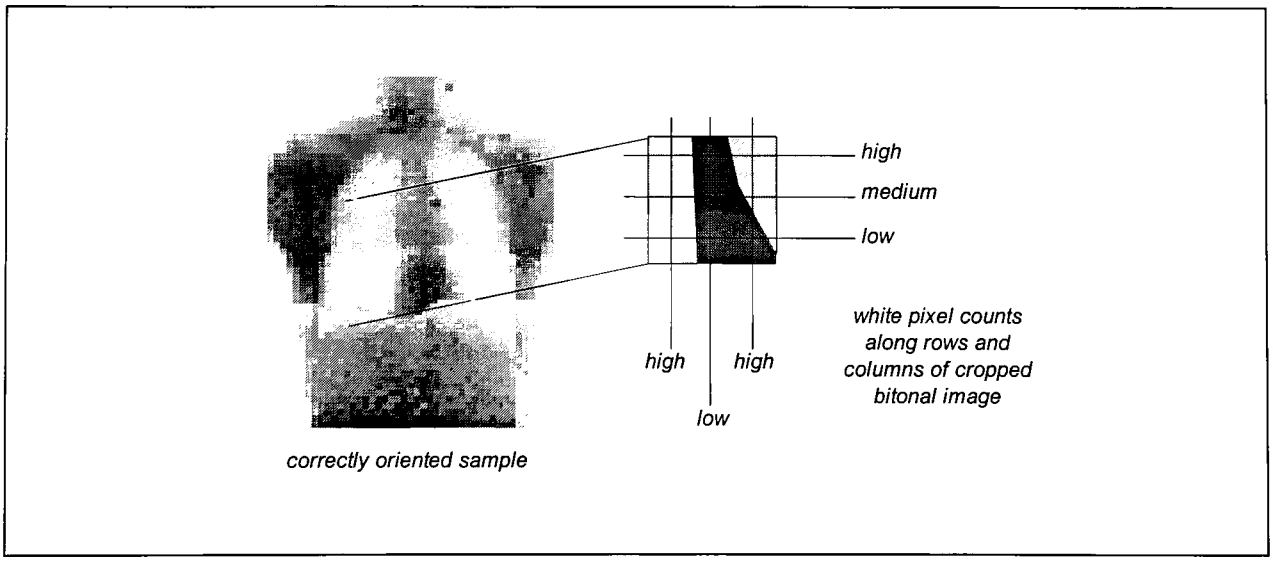
Unfortunately, a single threshold value does not work equally well for all of the sample images. Instead, a way is needed to compute an appropriate threshold value from the image data itself. To compute the threshold value, the feature extraction programs measure the average intensity values of pixels within a central sub-rectangle of the cropped image. The width and height of this region is selected to be half of the width (and height) of the cropped image.

3.2. Feature Set F_1

Both feature extraction programs work from the cropped, bitonal image (although these images are of different sizes in the two cases). Each feature value of feature set F_1 is simply a count of the number of white pixels in one of the rows or columns of the image. A count is computed for every other row, resulting in 10 feature values. Similarly, the other 10 feature values correspond to counts from every other column of the image.

The rationale for this particular feature set is the observation that in a correctly oriented image, the count of white pixels in a column of the cropped image is expected to be high when the column intersects a lung and low when the column lies between the lungs. Meanwhile, the count of white pixels in a row of the cropped, bitonal image is expected to be high near the top of the image, and lower nearer the bottom of the image, where the lungs may be obscured by other organs and by the presence of fluid in the lungs.

Figure 18: Rationale For Feature Set F_1



In theory, a high-low-high pattern (of white pixel counts) along the column-based features indicates that the orientation is correct or it is off by 180 degrees, while this same pattern along the row-based features would indicate

that the orientation is off by an odd multiple of 90 degrees. In either case, the values of the other 10 features can be used to select between the two candidate orientations.

Feature set F_1 is fairly efficient to compute. The threshold value is found by averaging the intensities of only 100 pixels of the image, and the threshold operation need not be explicitly performed. Instead of counting white pixels in the bitonal image, the feature extractor can count pixels in the original image whose intensities are greater than or equal to the threshold value. This test need only be performed over 10 rows and 10 columns of the cropped image, for a total of 400 pixels.

Unfortunately, this feature set considers only a relatively small portion of the image. One can expect considerable sensitivity to the position of the patient within the x-ray image. The second feature set designed by the author, feature set F_2, examines a slightly larger portion of the sample image and can therefore be expected to be somewhat less sensitive in this respect.

3.3. Feature Set F_2

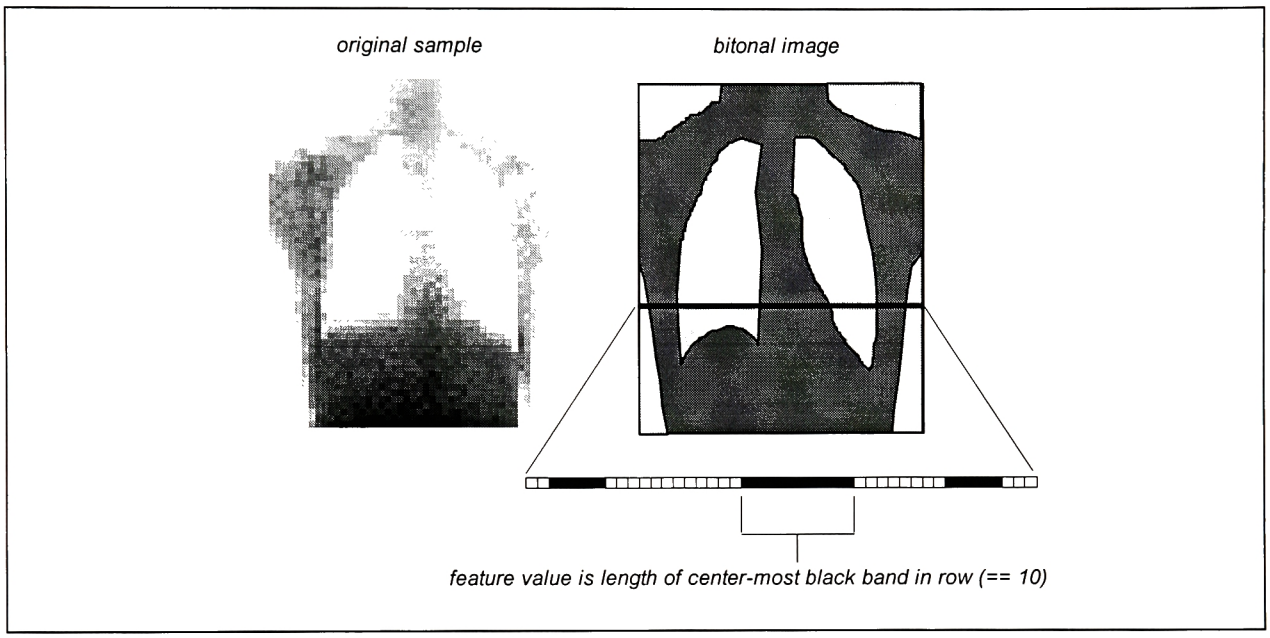
While feature set F_1 was designed to discriminate based on the position and orientation of the lungs, the design of feature set F_2 focuses on the dark area which lies between the lungs. As with the first feature set, F_2 computes a single feature value for each of 10 rows and 10 columns of the cropped image. In this case, however, the feature value is not a count of white pixels. Instead, it is the number of contiguous black pixels which comprise the center-most run of black pixels in the row or column. That is, each row or column of the bitonal image can be viewed as a series of alternating bands of white and black pixels. If there are one or more bands of black pixels, the feature is equal the number of pixels in the black band which lies closest to the center of that row or column.

Sometimes, the black band which is closest to the center of the row or column extends very close to the edge of the sample image. In such cases, the band is not likely to coincide with a horizontal cross-section of the dark area between the lungs, because a light band (which would represent a lung) is not present on one or both sides. The feature extraction program employs the following heuristics to maintain the focus of attention on the area between the lungs:

- If all pixels in the row or column are white, the feature value is zero.
- If the center-most black band extends to within three pixels of the edge of the image, then the feature value is equal to the total number of pixels in the row or column.

- If two black bands are equidistant from the center of the row or column, then the feature value is based on the longer of the two bands, unless that band extends to within three pixels of the edge (in which case the feature value is based on the shorter band)

Figure 19: Example Of A Feature Value For Feature Set F_2



These rules are designed to push the feature value to one of its extremes, in cases where there is doubt as to whether the band of black pixels actually represents a horizontal cross-section of the area between the lungs. In a correctly oriented image, most of the feature values for column-based features will assume one of these extreme values, while feature values for row-based features will lie somewhere between these extremes.

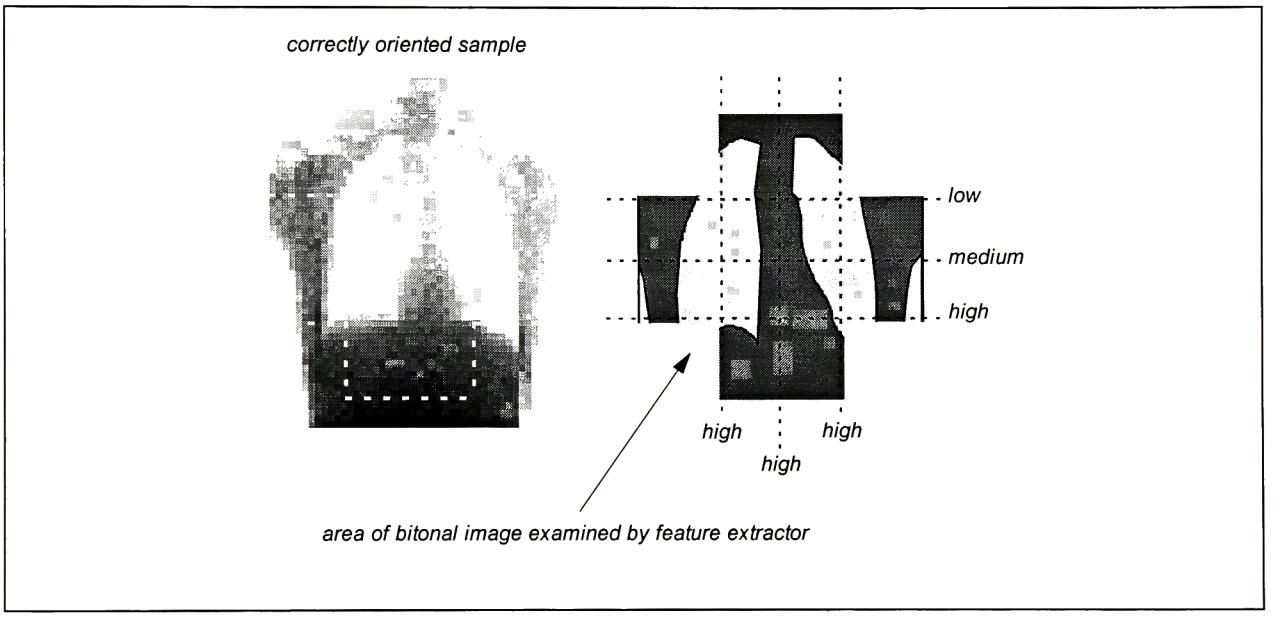
To further eliminate noise from the feature set, *line-filling* is performed along the row or column, before the feature value is measured. Line-filling involves removing isolated pixels of one color by replacing them with the other color. In the author's program, the row or column is scanned for bands consisting of only one or two white pixels. These are replaced by black pixels and the row or column is scanned a second time, looking for and replacing (with white) bands which consist of only one or two black pixels. Line-filling is done on a copy of the row or column, so as to not affect the data which is used to collect other feature values for the sample.

The rows and columns which are scanned to produce feature values correspond to the same rows and columns of the original sample image that were scanned for feature set F_1. However, for feature set F_2, each row and column is 45

pixels in length. Thus, as illustrated in Figure 20, a greater area of the image is considered for this feature set than for the previous feature set.

For a correctly oriented image, one would expect the values of each column-based feature to lie near one of the extremes of possible feature values (zero or 45). For row-based features, one would expect lower values near the top of the image and higher values near the bottom of the image.

Figure 20: Rationale For Feature Set F_2



Once again, the group of features extracted from what are actually columns of the correctly oriented image are expected to be quite distinguishable from the group of features extracted from what are the rows of that image. This difference should be sufficient to enable the classifier to narrow the decision to two choices (either zero and 180 degrees or 90 and 270 degrees), at which point the values of what are row-based features in the correctly oriented image can be compared to determine the exact class to which the sample should be assigned.

3.4. The Feature Extraction Program

Both feature extraction programs are implemented by a single C++ program whose behavior is tailored to one of the two feature sets by means of a compile-time switch. The program includes a class, *Image*, whose instances represent gray-scale images in memory. The constructor for the class reads a sample image from a file in the public domain, PGM format. *Image* implements the basic image processing operations described above, including

cropping, computation of the mean value of the *Image*, and the threshold operation. It also provides a method to rotate the image, but this method is not used by the feature extraction programs.

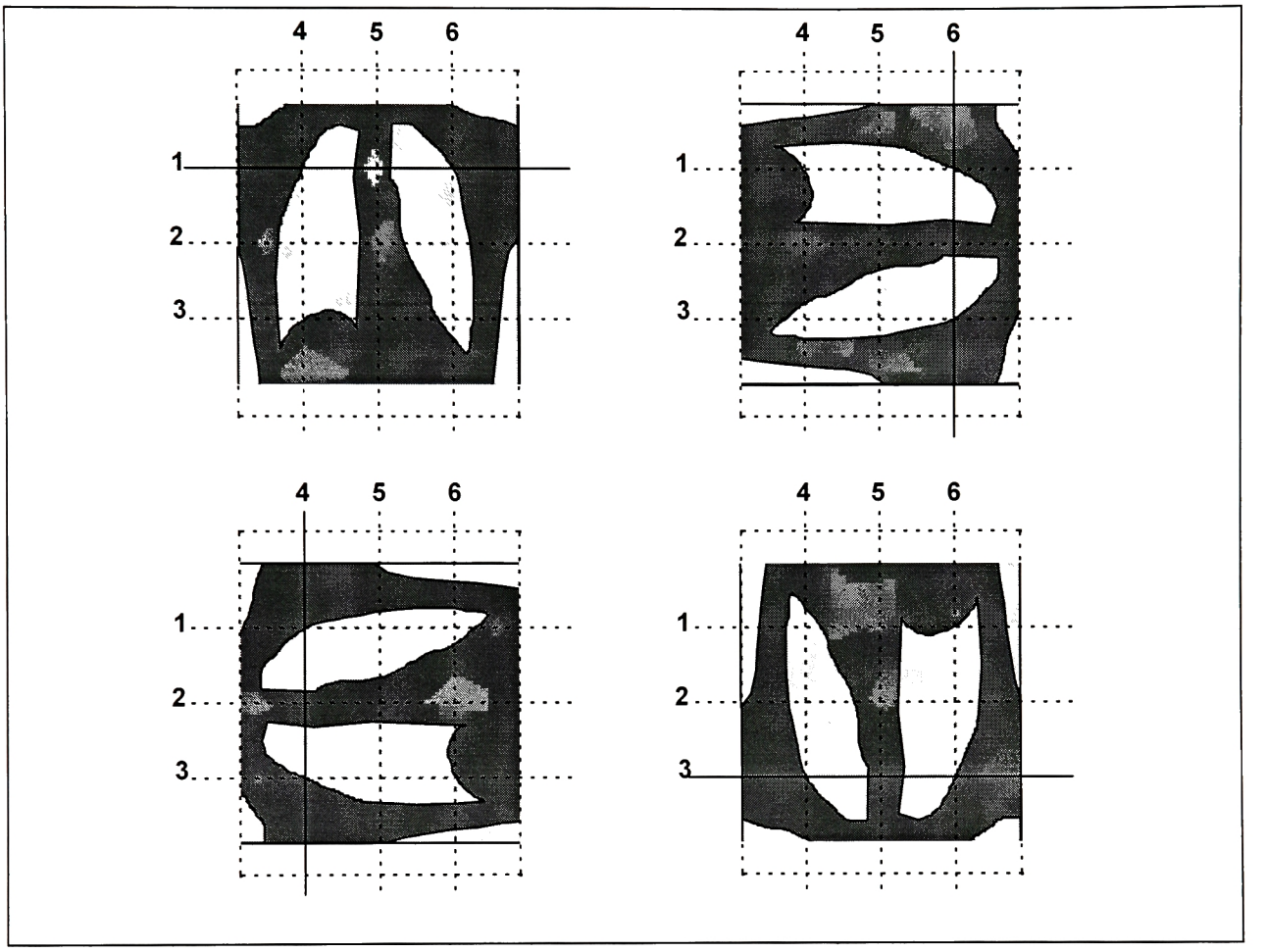
Each of the 20 features is represented by an instance of class *Feature1* (for feature set F_1) or *Feature2* (for feature set F_2). Each instance of either class is characterized by a straight-line "path" through the image, which is defined in terms of a starting (x, y) location, increment values for x and y, and the length of the path (i.e., the number of times x and y must be incremented to get from the start of the path to its end). Instances of *Feature1* simply count the number of white pixels which lie along the path. Instances of *Feature2* perform line-filling and then compute the length of the center-most black band of the path (applying the heuristics described earlier).

The main program reads a configuration file which provides the name of each sample image and its true orientation. For each sample, the program crops the image, computes the threshold value, and performs the threshold operation. The 20 feature objects are then asked to compute their values against the bitonal image which resulted from the threshold operation.

These 20 feature values, taken in order, correspond to the feature vector for the sample image, as it is oriented in the source image file. In fact, each sample image could be rotate through each of the four orientations, to produce four different feature vectors (thereby increasing the number of training exemplars by a factor of four).

Care was taken during the design of the feature sets so that each feature value in one orientation of the sample image will exactly correspond to another feature value in each of the other three orientations of that sample. Therefore, it is not necessary to actually rotate the image and measure the feature values for the other orientations. Each feature vector for these other orientations is simply a well-defined permutation of the 20 feature values already determined. For each sample image, the program writes four feature vectors to stdout, along with flags which indicate to which of the four classes of orientation each of the four feature vectors belongs.

Figure 21: Mapping Of A Single Feature Value In Each Of The Four Orientations



Together the 238 sample images provided to the author were converted into 952 training samples. With duplicate feature vectors removed, there were 936 samples for feature set F_1 and 924 samples for feature set F_2. These collections of feature vectors were saved to disk and were used in the various classification experiments which will be described subsequently.

4. Design Of Experiments

Three of the classification techniques studied are characterized by a number of parameters, the values of which must be defined by the experimenter as part of the design of his experiment. These parameters are documented in Tables 2, 3, and 4, for the Genetic Programming, Artificial Neural Network, and Probabilistic Neural Network techniques respectively. In contrast, the performance of the Linear Classifier technique is completely determined by the training data.

In addition to comparing the effectiveness and efficiency of the various techniques relative to one another, the author also attempted to measure the sensitivity of the first three techniques to the setting of these parameters. The selection of values for the parameters which characterize a technique is sometimes difficult to relate back to the problem being solved. Even within the context of the chest x-ray orientation problem, it is necessary to perform a suite of experiments with each technique, in order to present a fair comparison of them. By carefully designing these suites of experiments, the author was able to collect sensitivity data in conjunction with the performance data.

Unfortunately, the number of parameters which characterize a Genetic Programming experiment, makes an exhaustive study of its sensitivity to these parameters impractical. Instead, the author chose to examine just three of these parameters: population size, cross-over probability, and greedy over-selection threshold. The author also chose to study three parameters for the Artificial Neural Network and PNN techniques, as listed in Table 5.

Table 5: Parameters Selected For Sensitivity Analysis

```
Genetic Programming:
        Population Size
        Cross-Over Probability
        Greedy Over-Selection Threshold

Artificial Neural Network:
        Processing Units In Hidden Layer
        Learning Rate
        Momentum Factor

Probabilistic Neural Network:
        Number Of Stored Exemplars Per Class
        Learning Rate
        Smoothing Parameter
```

The decision to limit the study of sensitivity to only three parameters for each technique not only limits the scope of the experiments to something manageable, but it also allows the experiments to be designed using a method known as *Latin Square* design (Johnson, 1977:725).

4.1. The Latin Square Technique

The Latin Square technique allows for the separate analysis of three experimental parameters, studied at n levels (i.e., n different values) each, in only n^2 experiments. An exhaustive study of the combination of these parameters would require n^3 experiments. What is lost with the Latin Square is the ability to quantify interactions between pairs of the different parameters. These interactions, if present, increase the standard deviation of the experimental results (Johnson, 1977:727).

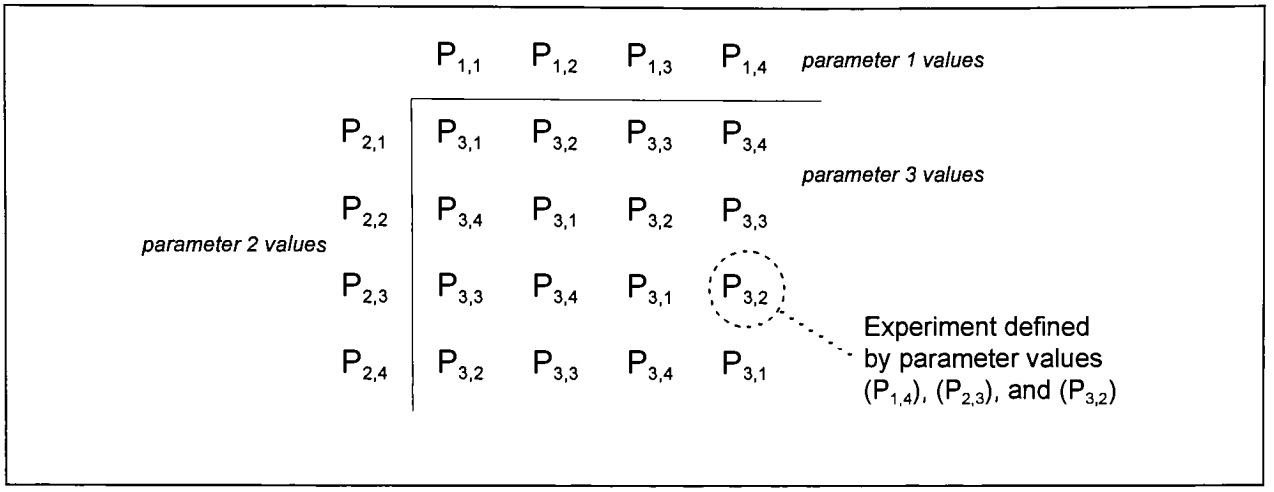
For the author's purposes, the Latin Square technique leads to an efficient design of experiments to exercise each of the techniques under a varying set of initial conditions and provides a direct measure of the relative sensitivity of the pattern classification technique to each of the three parameters.

Once the three parameters being studied have been identified, the other parameters which characterize the classification technique are fixed at reasonable, constant values. A single experiment is completely characterized then by the values assigned to the three parameters.

The Latin Square technique defines the n^2 experiments which need to be performed via an $n \times n$ matrix. Each element in the matrix defines one experiment. The column in which the element appears determines the value of the first parameter. Similarly, the element's row determines the value of the second parameter. Finally, the entry at the intersection of the row and column contains the value of the third parameter, completing the definition of one experiment.

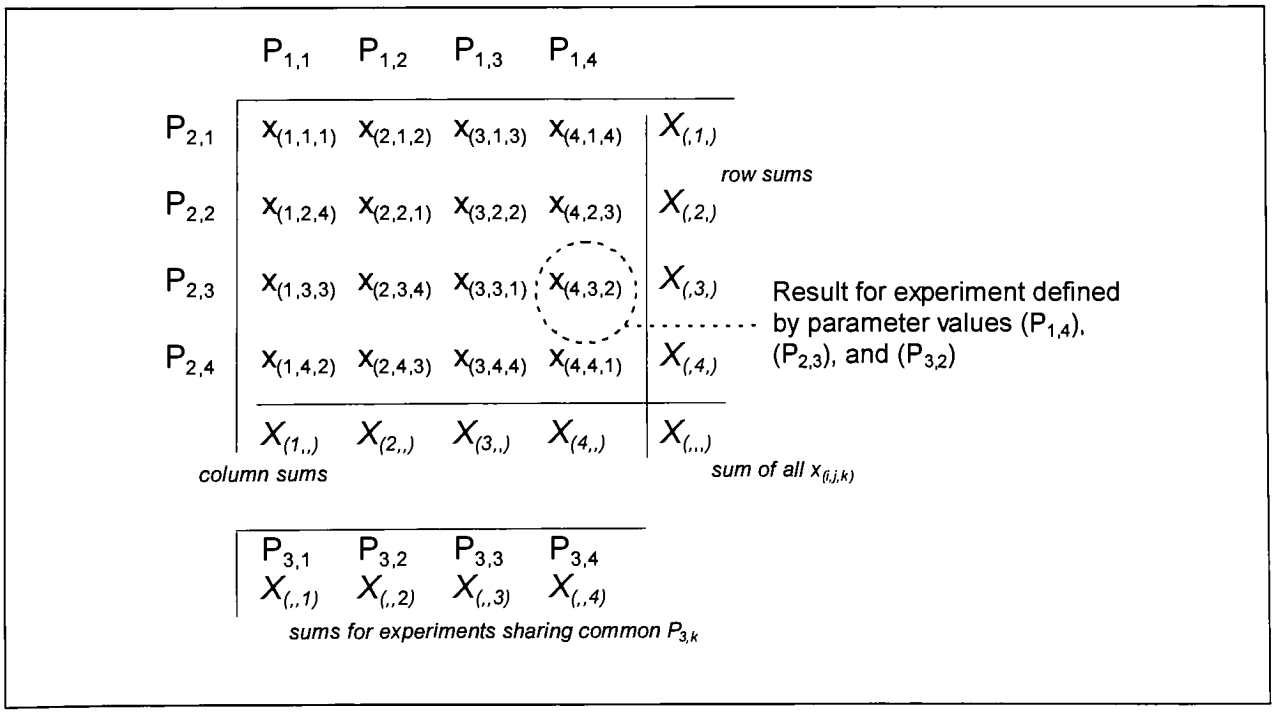
In Figure 22, the 4 x 4 matrix which lies below and to the right of the solid lines defines 16 different experiments. The order of the entries for values of parameter 3 does not need to match exactly the pattern presented in the figure. However, it is critical that each unique value for parameter 3 appear in exactly one row and exactly one column of the matrix (Johnson, 1977:726). The labels, $P_{i,j}$ used to represent the parameter values are not meant to imply that the values are sorted. Indeed, it has been suggested that the labels be assigned randomly to the n different values of parameter i, to eliminate bias (Finney, 1960:48).

Figure 22: Example Of A 4 x 4 Latin Square Design



The results of conducting the n^2 experiments defined in the Latin Square can be presented in a matrix, where each element of the matrix contains the result of conducting the experiment defined by the corresponding element of the Latin Square. Of interest is the sum of the results of all n^2 experiments, as are the sums of results for each group of n experiments which share a common value for one of the parameters. Figure 24 illustrates the results of a 4 x 4 Latin Square of experiments, including the $(3n+1) = 13$ sums.

Figure 23: A Presentation Of Results From A Latin Square Experiment



4.2. Analysis Of Variance

Table 6 presents the analysis of variance for the Latin Square design (Johnson, 1977:729). The "sum of squares", S_1, is the variance of the column sums. Similarly, S_2 is the variance of the row sums and S_3 is the variance of the sums of results which share common values of parameter 3.

Table 6: Analysis Of Variance For Latin Square Design

	Sum Of Squares	D.F.	Mean Square	M.S. Ratio
Param 1:	$S_1 = n^{-1}\sum_i X^2_{(i,,)} - X^2_{(,,,)}/n^2$	$n-1$	$S_1/(n-1)$	$\frac{S_1(n-2)}{S_e}$
Param 2:	$S_2 = n^{-1}\sum_j X^2_{(,j,)} - X^2_{(,,,)}/n^2$	$n-1$	$S_2/(n-1)$	$\frac{S_2(n-2)}{S_e}$
Param 3:	$S_3 = n^{-1}\sum_k X^2_{(,,k)} - X^2_{(,,,)}/n^2$	$n-1$	$S_3/(n-1)$	$\frac{S_3(n-2)}{S_e}$
Residual:	$S_e = S - S_1 - S_2 - S_3$	$(n-1)(n-2)$	$\frac{Se}{(n-1)(n-2)}$	
Total:	$S = \sum_i \sum_j x^2_{(i,j,k)} - X^2_{(,,,)}/n^2$	n^2-1		

D.F.= "degrees of freedom"
M.S. = "mean square"

One expects higher values for the "sum of squares" corresponding to parameters to which the experiment is more sensitive. Thus, the experiment's relative sensitivity to the three parameters can be determined by direct comparison of these values. However, it is the value of the mean square ratio which is used to determine the statistical significance of the observed sensitivity.

If one assumes the *normal error model*, i.e., that the variation in results due to uncontrolled elements of the experiment is normally distributed with a mean of zero, it can be shown that the mean square ratios in the last column of Table 6 follow the standard F-distribution for variance ratios (Finney, 1960:20). Thus, if the mean square ratio for one of the parameters lies within the 95% confidence interval of the standard F-distribution with $(n-1)$ degrees of freedom in the numerator and $(n-1)(n-2)$ degrees of freedom in the denominator, then the corresponding parameter (within the range of values tested) is considered to be a significant factor affecting the

results of the experiment. For the experiments conducted by the author, this corresponds to a mean square ratio greater than or equal to 4.76.

In the design of a Latin Square, consideration must be given to the number of levels against which each of the different parameters will be tested (i.e., the value for n). Intuitively a large value of n is likely to produce results from which conclusions can be drawn with greater confidence. However, larger values for n increase the cost of performing the experiments and analyzing the results.

The conclusions which will be drawn from this set of experiments are specific to the chest x-ray orientation problem and cannot be responsibly elevated to the scope of pattern classification problems in general. Although, in the end, the author will appeal to common sense in arguing that something has been learned about the relative merits of the four different techniques, beyond their application to the specific problem studied, increasing the number of experiments performed in each of the Latin Squares would in no way strengthen these arguments.

Thus, in each case, the author has chosen to use a 4×4 Latin Square. It is his belief that the 16 experiments associated with each Square are sufficient to draw conclusions about the performance of the associated classification technique with a reasonable degree of confidence. Given the relatively controlled nature of the experiments, the author hopes that this number of experiments will also be sufficient to legitimize the analysis of variance which will be performed on the results.

5. The Chest X-Ray Orientation Experiments

This section describes the experiments which were performed with each of the pattern classification techniques described earlier, as well as the results of those experiments.

5.1. Genetic Programming Experiments

As mentioned earlier, a suite of Genetic Programming experiments was designed using a Latin Square for three of the parameters: population size, cross-over probability, and greedy over-selection threshold. This last parameter is the fraction of total fitness allocated to "group 1" during the greedy over-selection process.

Figure 24: Latin Square Used In Genetic Programming Experiments

Greedy over-selection threshold	cross-over probability 0.85	0.90	0.95	0.99
0.20	1000	500	2000	1500
0.30	1500	1000	500	2000
0.50	2000	1500	1000	500
0.40	500	2000	1500	1000

(population size)

Table 7 lists other parameters which characterize the Genetic Programming technique and the values used in the author's experiments.

Mutation, permutation, edit, and encapsulation operations were either not implemented or were disabled. Decimation of the population during early generations was also disabled. The probability of selecting a particular node as a cross-over point was independent of node type.

Table 7: Fixed Parameters In The Genetic Programming Experiments

Terminal Set:	20 feature values plus constants 0, 1, 2, & 3
Function Set:	(int) *add, greater-than, and, or, not, & if*
Fitness Measure:	number of training samples correctly oriented
Max. Generations:	75
Max. Initial Tree Depth:	6
Max. Tree Depth:	17
Prob. Asexual Rep:	(1.0 - cross-over probability)
Prob. Group-1 Selection:	0.80

Naturally, the selection of an appropriate *function set* is critical to the success of the Genetic Programming experiment. In this instance, the author felt confident that the function set described in Table 7 would be sufficient, because he had already implemented a traditional program for classifying the samples, which was quite successful and could be implemented as parse tree built from this function set. The author's classifier program will be described later in the paper.

The class to which a sample is assigned is determined by the value of the low order two bits of the result of evaluating a parse tree. The fitness of a particular program is simply equal to the number of correct responses to the training data. The four constants 0, 1, 2, and 3 were included as inputs, so that the generated programs could easily return values corresponding to all four orientations.

In each experiment, the first 60% of the sample data was used for training. The other 40% was used to measure the classifiers ability to generalize. The suite of 16 experiments defined by the Latin Square in Figure 24 was executed for each of the two feature sets described earlier in the paper.

A *best-of-generation* program is defined to be a program which correctly classifies more of the samples used for training than does any other program in its generation for a particular run of the simulator. The simulator implemented by the author tests each best-of-generation program against the 40% of the sample data which was not used for training. The *best-of-run* program is defined to be the *best-of-generation* program which correctly classified more of the combined samples than did any of the other tested programs within the run.

Table 8 summarizes the results of the 16 runs of the Genetic Programming simulator for feature set F_1. The first column of the table associates a run identifier with each of the 16 runs. The next three columns identify the values of the three parameters from Figure 24.

Table 8: Results Of Genetic Programming Experiments Against Feature Set F_1

					BEST-OF-RUN				
RUN	XOVER	GREEDY	POPSZ	TOT-CPU	GEN	%TRAIN	%TEST	%TOTAL	CPU
1	0.85	0.20	1000	1363	33	98.0	91.2	95.3	581
2	0.90	0.20	500	615	49	74.9	71.7	73.6	375
3	0.95	0.20	2000	2688	45	99.3	95.7	97.9	1377
4	**0.99**	**0.20**	**1500**	**3246**	**50**	**98.6**	**97.3**	**98.1**	**2462**
5	0.85	0.30	1500	3133	67	98.8	93.3	96.6	2758
6	0.90	0.30	1000	1993	67	98.4	93.3	96.4	1753
7	0.95	0.30	500	715	41	74.3	73.9	74.1	375
8	0.99	0.30	2000	2470	43	74.3	73.3	73.9	1407
9	0.85	0.50	2000	2314	64	97.0	95.2	96.3	1880
10	0.90	0.50	1500	2674	73	95.2	90.4	93.3	2579
11	0.95	0.50	1000	2162	70	97.5	92.5	95.5	1979
12	0.99	0.50	500	783	61	74.5	73.3	74.0	600
13	0.85	0.40	500	1017	68	97.3	95.7	96.7	919
14	0.90	0.40	2000	2104	74	96.6	89.9	93.9	2066
15	0.95	0.40	1500	1978	71	97.0	90.4	94.3	1863
16	0.99	0.40	1000	2491	57	98.4	92.5	96.0	1641
				------	--	------	------	------	----
			MEAN	1984	58	91.9	88.1	90.4	1538

The column labeled "TOT-CPU" reports the total CPU usage (in seconds) for all 75 generations in each run of the simulator. The columns labeled "BEST-OF-RUN" report statistics on the *best-of-run* program for each run. The statistics reported include (in order): the generation number of the *best-of-run* program, the accuracy of the program at classifying training samples and test samples, the weighted average of these two percentages, and the cumulative CPU time (in seconds) used for generations up to and including the generation which resulted in the *best-of-run* program.

The overall best program of the 16 runs of the simulator for feature set F_1 was produced in generation 50 of run number 4. That program correctly classifies 98.1% of all samples, including 97.3% of the samples which were not used for training. Other programs were found to be better at classifying the samples used for training, but no other program correctly classified more of the blind test samples. This "overall best" program is shown in Figure 25 as a LISP-like expression. The parse tree consists of only 48 nodes, with a maximum depth of 9.

Figure 25: Overall Best Program For Classification Based On Feature Set F_1

```
(& (| (> (> 2 (| (> (> i10 i17) i1) (+ (| 1 (> i20 i13)) (> i15 i11)))) i15)
      (+ (| (| 1 (> i2 (? i13 i8 i10))) (> i15 i11)) (> i15 i11)))
   (| (& (> i20 i17) (| 1 3)) (> i19 i14)))
```

Interestingly, this program references only four of the 10 column-based features and only seven of the 10 row-based features. The program was created in generation 50 of its run, which is only two-thirds of the way toward the run-limit of 75 generations. The 75th generation produced a smaller program which correctly classified the exact same number of samples. This program consists of only 21 nodes in a parse tree whose maximum depth is six. This program is shown in Figure 26.

Figure 26: Program Which Tied The Overall Best Program For Feature Set F_1

```
(& (+ (| (| 1 (> i2 i8)) (> i15 i11)) (> i15 i11)) (| (> i20 i17) (> i19 i14)))
```

The program in Figure 26 references only 2 of the column-based features and 6 of the row-based features. A detailed analysis of the way this program classifies the orientation of chest x-rays is included in Appendix A.

The run which produced these overall best classifiers required 3,246 seconds of CPU time out of a total of 8.8 hours of CPU time for all 16 runs. The average overall accuracy of a program produced by a single run was 90.4%. The average run required 1,984 seconds of CPU time.

The analysis of variance in Table 9 shows that the accuracy of the best-of-run program was most sensitive to population size (of the parameters studied and in the ranges tested). However, the mean square ratio of 4.07 is less than $F_{3,6,0.95} = 4.76$, and one cannot conclude that changes to any one of the three parameters by itself has a statistically significant affect on the results.

Table 9: Analysis Of Variance For "% Total Correct" From Table 8

SOURCE	|------- SUMS --------|				SUM OF SQUARES	D.F.	MEAN SQUARE	M.S. Ratio
X-over:	384.8	357.2	361.9	342.1*	234.95	3	78.32	1.39
Greedy:	364.9	341.0	381.0	359.1	203.96	3	67.99	1.21
Pop Sz:	318.5	383.2	382.3	362.0	688.48	3	229.49	4.07
Residual:					338.01	6	56.34	
Total:				1445.9	1465.41	15		

**sums in aov tables are listed in order of increasing parameter value*

Despite this result, one can see from Table 8 that three out of the four runs of the simulator with a population size of 500 resulted in best-of-run programs with an accuracy below 75%. In contrast, the runs with a population size of 1,500 always resulted in a program which scored above 93%.

As one might expect, population size does have a statistically significant affect on the CPU time required to execute a complete run of the simulator (see Table 10). Perhaps less obvious is the sensitivity of the simulator to the greedy over-selection threshold, which appears to affect the rate at which the simulator converges on a good solution (see Table 11).

Table 10: Analysis Of Variance For Total CPU Time From Table 8

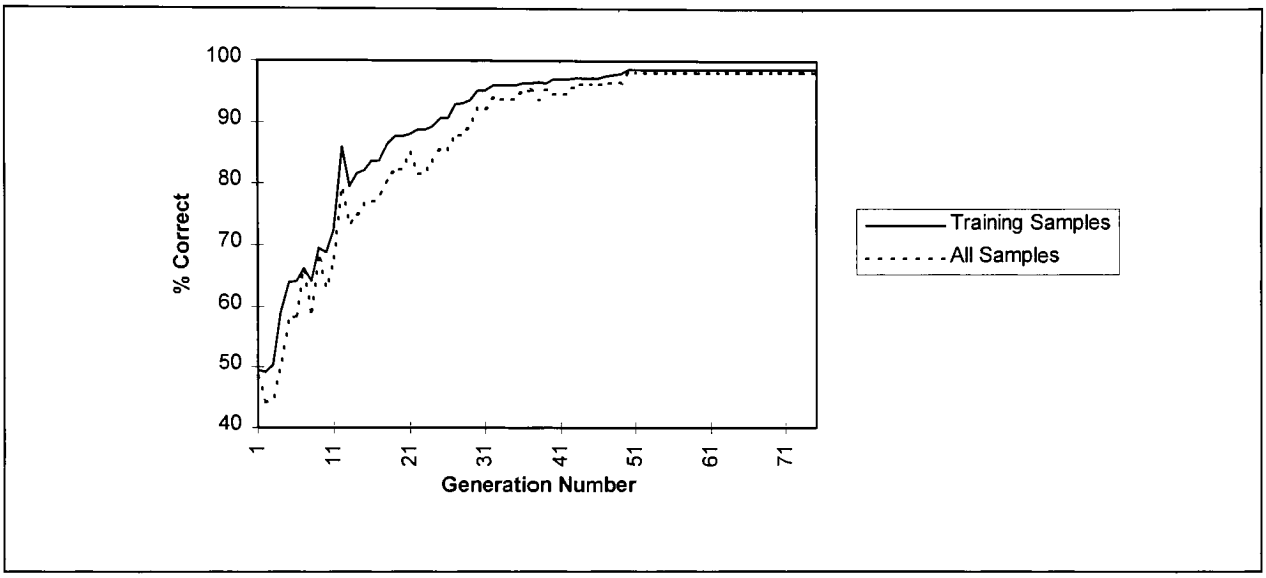
SOURCE	|------- SUMS -------|				SUM OF SQUARES	D.F.	MEAN SQUARE	M.S. Ratio
X-over:	7829	7387	7545	8991	394657.28	3	131552.43	0.53
Greedy:	7914	8312	7591	7934	65354.33	3	21784.78	0.09
Pop Sz:	3131	8011	11032	9577	8843021.21	3	2947673.74	**11.98**
Residual:					1476422.92	6	246070.49	
Total:				31753	10779455.73	15		

Table 11: Analysis Of Variance For Best-Of-Run Generation Number From Table 8

SOURCE	|------- SUMS ------|				SUM OF SQUARES	D.F.	MEAN SQUARE	M.S. Ratio
X-over:	232	263	227	211	355.19	3	118.40	1.60
Greedy:	177	218	270	268	1488.69	3	496.23	**6.72**
Pop Sz:	219	227	261	226	266.19	3	88.73	1.20
Residual:					443.38	6	73.90	
Total:				933	2553.44	15		

Figure 27 summarizes the performance of run number four of the Genetic Programming simulator, which produced the overall best classifier for feature set F_1. Shown are the percentage of correctly classified training samples and correctly classified samples overall, for each of the best-of-generation programs in this run.

Figure 27: Classifier Accuracy By Generation For Best Genetic Programming Run Against F_1



As one might expect, the differences in performance between successive generations is much more significant in earlier generations. Once the simulator achieved 95% accuracy against the training data (generation 30), the rate of improvement in accuracy declined significantly. After generation 50, no improvement in classifier accuracy is observed.

Overall, the programs which result from runs of the Genetic Program simulator against feature set F_2 perform somewhat better than those of the runs against feature set F_1. However, the parsimony which characterizes the program in Figure 26 will be seen to be noticeably absent. The results for the runs of the simulator against this second feature set are presented in Table 12.

Table 12: Results Of Genetic Programming Experiments Against Feature Set F_2

					BEST-OF-RUN			
RUN	XOVER	GREEDY	POPSZ	TOT-CPU	GEN	%TRAIN	%TEST	%TOTAL	CPU
1	**0.85**	**0.20**	**1000**	**2234**	**61**	**99.8**	**97.0**	**98.7**	**1775**
2	0.90	0.20	500	1001	67	85.4	78.9	82.8	906
3	0.95	0.20	2000	2838	33	98.4	93.2	96.3	1230
4	0.99	0.20	1500	1891	70	99.6	96.8	98.5	1762
5	0.85	0.30	1500	1837	64	99.3	94.9	97.5	1592
6	0.90	0.30	1000	1199	45	98.9	95.1	97.4	677
7	0.95	0.30	500	850	69	97.7	92.2	95.5	759
8	0.99	0.30	2000	3782	67	99.6	94.3	97.5	3331
9	0.85	0.50	2000	3477	53	98.6	94.1	96.8	2200
10	0.90	0.50	1500	2099	40	90.4	82.2	87.1	759
11	0.95	0.50	1000	1812	73	98.9	95.7	97.6	1737
12	0.99	0.50	500	835	75	95.7	89.5	93.2	835
13	0.85	0.40	500	887	72	91.9	82.4	88.1	852
14	0.90	0.40	2000	3390	36	98.4	93.8	96.5	1436
15	0.95	0.40	1500	3324	74	98.6	94.1	96.8	3261
16	0.99	0.40	1000	1757	70	98.6	97.8	98.3	1661
				------	--	------	------	------	----
			MEAN	2076	60	96.9	92.0	94.9	1548

In this set of experiments, the overall best program was produced in generation 61 of run number one. The program consists of 194 nodes with a maximum tree depth of 15, and it correctly classifies 98.7% of all samples, including 97% of samples which were not used during training.

As in the previous case, the simulator produced a smaller program (92 nodes) with the same performance, in a later generation of the same run. This program is shown in Figure 28.

Figure 28: Overall Best Program For Classification Based On Feature Set F_2

```
(| (& (& (> (? i8 (? i3 i15 3) (~ i8)) i18) (~ 1)) (| i13 (> i5 i17)))
   (| (? (> i2 i7)
         (? i3
            (? (> i12 (& (> i12 i18) (& i6 (> (+ i2 i6) i5))))
               (? (? (? i15 i6 (> i2 i7)) i17 i3) i17 i19)
               (~ 1))
            3)
         (> i15 i5))
      (? i7
         (? (~ (> i12 (| i19 i20)))
            (& (> i9 i7)
               (| (? (> (| i5 (> i5 i17)) i7) i12 (> i6 i5))
                  (? i3 i17 3)))
            (~ i2))
         i3)))
```

Although the program in Figure 28 exhibits an overall performance which is better than the best program for feature set F_1, the program's performance when used to classify samples which were not used during training is slightly worse (there is some danger in making these comparisons, because the number of samples, after eliminating duplicates, are slightly different for the two feature sets). The *best-of-run program* for run 16 performs poorer overall, but correctly classified 97.8% of the samples which were not used for training (see Figure 29).

Figure 29: Best, Best-Of-Run Program For Classifying Non-Training Samples Of F_2

```
(| (> i8 i4)
   (? (& (& i7
             (> (> i14 (| i18 (> i14 (| 3 i16))))
                (| (> (& i7 i15) (~ i16)) i17)))
          (| i18 i20))
      (& (+ (> i8 i14) (| i10 i17)) 1)
      (~ (? 1
            i3
            (| (> i14
                  (> (? (& i14 (+ i3 i16)) i13 (& i16 (~ i17))) (~ i20)))
               (? i11
                  (& i10 (~ i10))
                  (~ i17)))))))
```

As with the analysis of variance for the other feature set, one cannot conclude that the overall accuracy of the classifiers produced for feature set F_2 exhibits a statistically significant sensitivity to the three tunable parameters (in the ranges of values tested). Table 13 summarizes this analysis. However, it is still worth noting that the runs which used a population size of 500 performed consistently worse than runs which used greater population sizes.

Table 13: Analysis Of Variance For "% Total Correct" From Table 12

SOURCE	|-------- SUMS -------|		SUM OF SQUARES	D.F.	MEAN SQUARE	M.S. Ratio
X-over:	381.1 363.9 386.1 387.4		88.63	3	29.54	2.67
Greedy:	376.3 387.9 379.7 374.7		25.92	3	8.64	0.78
Pop Sz:	359.5 392.0 379.9 387.1		153.31	3	51.10	4.62
Residual:			66.31	6	11.05	
Total:		1518.5	334.17	15		

As Table 14 shows, the total CPU time per run is highly sensitive to population size. This is consistent with intuition and with the results for the first feature set.

Table 14: Analysis Of Variance For Total CPU Time From Table 12

SOURCE	SUMS				SUM OF SQUARES	D.F.	MEAN SQUARE	M.S. Ratio
X-over:	8437	7691	8826	8267	166969.79	3	55656.60	0.17
Greedy:	7967	7670	9361	8224	409684.81	3	136561.60	0.43
Pop Sz:	3575	7004	9153	13489	12916417.95	3	4305472.65	**13.48**
Residual:					1917076.20	6	319512.70	
Total:				33222	15410148.75	15		

Unlike the results for feature set F_1, the results for feature set F_2 do not show the same sensitivity of the convergence rate to the greedy over-selection threshold. Indeed, the analysis of variance in Table 15 shows just the opposite. For this feature set, the greedy over-selection threshold had almost no effect on the generation in which the best-of-run program was created. There appears to be some sensitivity to cross-over probability and population size, but none of the mean square ratios in Table 15 exceed the 95% confidence limit of 4.76, necessary to conclude statistical significance.

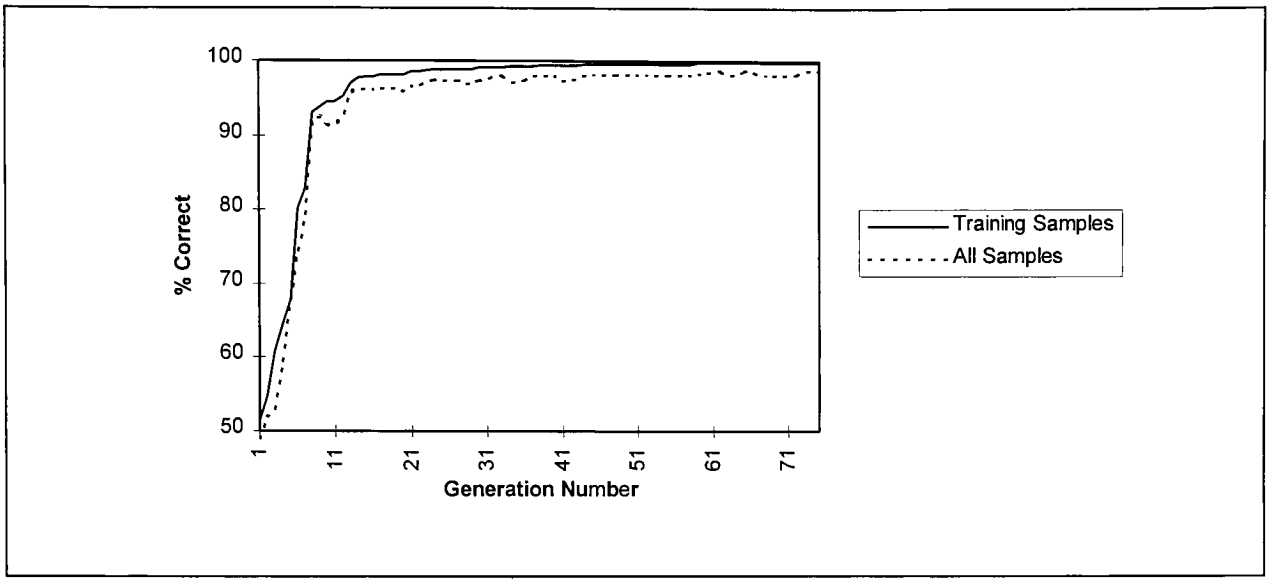
Table 15: Analysis Of Variance For Best-Of-Run Generation Number From Table 12

SOURCE	SUMS				SUM OF SQUARES	D.F.	MEAN SQUARE	M.S. Ratio
X-over:	250	188	249	282	1157.19	3	385.73	3.10
Greedy:	231	245	252	241	57.69	3	19.23	0.15
Pop Sz:	283	249	248	189	1143.69	3	381.23	3.07
Residual:					745.38	6	124.23	
Total:				969	3103.94	15		

The run which produced the overall best classifier for feature set F_2 required 2,234 seconds of CPU time out of a total of 9.2 hours of CPU time for all 16 runs. The average overall accuracy of a program produced by a single run was 94.9% (4.5% better than the average for feature set F_1). The average run in Table 12 required 2,076 seconds of CPU time.

Figure 30 summarizes the performance of run number one of the Genetic Programming simulator against feature set F_2. Once again, the accuracy of the best-of-generation programs for each generation are shown, as measured by the percent of training samples and percent of samples overall which the program classifies correctly.

Figure 30: Classifier Accuracy By Generation For Best Genetic Programming Run Against F_2



When this graph is compared against that of Figure 27, one sees that the simulator converged on a "good" solution much more quickly for feature set F_2. In addition, the percentage of training samples correctly classified increases monotonically in Figure 30, while in early generations for feature set F_1, the best-of-generation program sometimes performed more poorly than the best-of-generation program in earlier generations. Whether these differences are accounted for by variations in the three run parameters shown in Figure 24 or differences in "sophistication" of the two feature sets is difficult (if not impossible) to determine from the data available.

5.2. Artificial Neural Network Experiments

This section describes the chest x-ray orientation experiments conducted by the author using his feedforward artificial neural network simulator. Once again, the Latin Square technique was used to design a suite of 16 experiments, which were then performed against each of the two feature sets.

Figure 31: Latin Square Used In Artificial Neural Network Experiments

learning rate \ momentum factor	0.5	0.6	0.7	0.8
0.10	15	5	20	10
0.20	10	15	5	20
0.35	5	20	10	15
0.50	20	10	15	5

processing units in hidden layer

In each experiment, the network included one hidden layer, with a number of processing units determined by the experiment's position in the Latin Square shown in Figure 31. Weight adjustment was done "by sample", and the gain parameter, λ, was equal to one.

The inputs to the network were the 20 feature values extracted from a sample x-ray. An experiment consisted of 100 passes through the training data. As with the Genetic Programming experiments (and the experiments with the other classification techniques), the first 60% of the sample data was used for training. The remaining 40% was used to evaluate the network's success at generalization.

The author's neural network simulator normalizes each feature value of each *training* sample prior to start of the experiment. This normalization is based on the minimum and maximum observed value of each feature amongst all of the sample data. A feature value is converted to its normalized value using Equation 17. Table 16 lists the normalization constants that were used in Equation 17.1 for each of the 20 features in each of the two feature sets.

Equation 17: Formula Used To Normalize Feature Values

$$f_i' = (f_i - m_i) / r_i \qquad \text{(Equation 17.1)}$$

where f_i is the value for feature i from one of the samples,
m_i is a normalization constant equal to the minimum observed value of f_i amongst all samples used for training, and
r_i is a normalization constant equal to the difference between the maximum and minimum observed values of f_i amongst all samples used for training.

Table 16: Normalization Constants For Each Feature Value Of Each Feature Set

	F_1		F_2			F_1		F_2	
Feature	m_i	r_i	m_i	r_i	*Feature*	m_i	r_i	m_i	r_i
1	0.0	21.0	0.0	45.0	11	0.0	21.0	0.0	45.0
2	0.0	21.0	0.0	45.0	12	0.0	21.0	0.0	45.0
3	0.0	21.0	3.0	42.0	13	0.0	21.0	3.0	42.0
4	0.0	21.0	4.0	41.0	14	0.0	21.0	4.0	41.0
5	0.0	21.0	7.0	38.0	15	0.0	21.0	7.0	38.0
6	0.0	21.0	7.0	38.0	16	0.0	21.0	7.0	38.0
7	0.0	21.0	4.0	41.0	17	0.0	21.0	4.0	41.0
8	0.0	21.0	3.0	42.0	18	0.0	21.0	3.0	42.0
9	0.0	21.0	0.0	45.0	19	0.0	21.0	0.0	45.0
10	0.0	21.0	0.0	45.0	20	0.0	21.0	0.0	45.0

After every two training passes, the current network was evaluated against the 40% of samples not used for weight adjustment, and a snapshot of the resulting confusion matrices was logged. A *best-of-run* network is defined to be the network corresponding to the snapshot in which more of the combined samples (training and test) were correctly classified than in any other snapshot of the run.

Table 17 summarizes the performance of the 16 best-of-run networks for feature set F_1. There is much less variation amongst the results of these experiments than was observed for the Genetic Programming experiments. The CPU times in this table are in units of seconds, and the difference between these values and those of the Genetic Programming experiments (roughly two orders of magnitude) emphasize the relative efficiency of the neural network technique.

Table 17: Results Of Feedforward Neural Network Experiments Against Feature Set F_1

					BEST-OF-RUN				
RUN	MOMENT	LEARN	NODES	TOT-CPU	PASS	%TRAIN	%TEST	%TOTAL	CPU
1	0.50	0.10	15	45	96	100.00	96.27	98.50	43
2	0.60	0.10	5	18	96	99.82	96.00	98.29	17
3	0.70	0.10	20	60	90	100.00	96.27	98.50	54
4	0.80	0.10	10	32	76	100.00	96.80	98.72	24
5	0.50	0.20	10	32	68	100.00	96.80	98.72	21
6	0.60	0.20	15	45	76	100.00	96.53	98.61	34
7	0.70	0.20	5	18	64	100.00	97.07	98.82	11
8	0.80	0.20	20	60	68	100.00	95.47	98.18	41
9	0.50	0.35	5	18	68	100.00	96.80	98.72	12
10	0.60	0.35	20	60	44	99.82	95.47	98.08	26
11	0.70	0.35	10	32	92	100.00	97.07	98.82	29
12	**0.80**	**0.35**	**15**	**45**	**50**	**100.00**	**97.33**	**98.93**	**23**
13	0.50	0.50	20	60	56	100.00	95.47	98.18	33
14	0.60	0.50	10	32	56	99.82	97.33	98.82	17
15	0.70	0.50	15	45	46	100.00	97.07	98.82	21
16	0.80	0.50	5	18	92	99.82	96.27	98.40	16
			MEAN	39	71	99.96	96.50	98.57	26

In this set of experiments, the overall best network was the best-of-run network for run number 12, which correctly classified 98.93% of the samples, including all of the training samples and 97.33% of the samples not used to adjust weights during training. Only three out of the 936 samples were classified incorrectly. The classification results for seven other samples were judged ambiguous.

The network included 15 nodes in its hidden layer. The weights for these nodes and the four output layer nodes are listed in Appendix B. The entire experiment required 45 seconds of CPU time, but the network achieved 100% accuracy against the training data after only 22 seconds of CPU time (48 training passes). Had the experiment been stopped at that point, the resulting network would have scored exactly the same number of correct answers, but there would have been five incorrect responses and five which were ambiguous, instead of three and seven respectively.

Table 18 summarizes the analysis of variance for the overall accuracy of neural network classifiers for feature set F_1. Although the number of nodes in the hidden layer appears to be a significant source of variance in the results ($6.94 > F_{3,6,0.95} = 4.67$), one must temper this observation with the fact that the variance itself is minimal. Indeed, each of the 16 best-of-run networks correctly classify between 98 and 99 percent of all samples, with an average of 98.57%.

Table 18: Analysis Of Variance For "% Total Correct" From Table 17

SOURCE	|------- SUMS --------|				SUM OF SQUARES	D.F.	MEAN SQUARE	M.S. Ratio
Momentum:	394.1	393.8	395.0	394.2	0.19	3	0.06	1.86
Learn Rate:	394.0	394.3	394.2	394.6	0.04	3	0.01	0.37
Nodes:	394.2	395.1	394.9	392.9	0.69	3	0.23	**6.94**
Residual:					0.20	6	0.03	
Total:				1577.1	1.12	15		

As the analysis of variance in Table 19 shows, the rate at which the simulator arrived at the best-of-run classifier does not appear to be sensitive to any of the three parameters by themselves (within the ranges of values tested). In this case however, the variance between individual experiments is quite high. The relatively high residual variance shown in Table 19 is probably a result of sensitivity to interactions between the three variables.

Table 19: Analysis Of Variance For Best-Of-Run Training Pass Number From Table 17

SOURCE	|------ SUMS -------|				SUM OF SQUARES	D.F.	MEAN SQUARE	M.S. Ratio
Momentum:	288	272	292	284	56.00	3	18.67	0.05
Learn Rate:	358	276	250	252	1930.00	3	643.33	1.62
Nodes:	320	292	266	258	590.00	3	196.67	0.50
Residual:					2376.00	6	396.00	
Total:				1136	4952.00	15		

As one might expect, the amount of CPU time required by one run of the neural network simulator is highly sensitive to the number of nodes in the hidden layer of the network. Any doubt as to the statistical significance of this sensitivity is alleviated by the analysis of variance in Table 20. The fact that the values for momentum and learning rate do not appear to significantly affect CPU usage is also consistent with common sense.

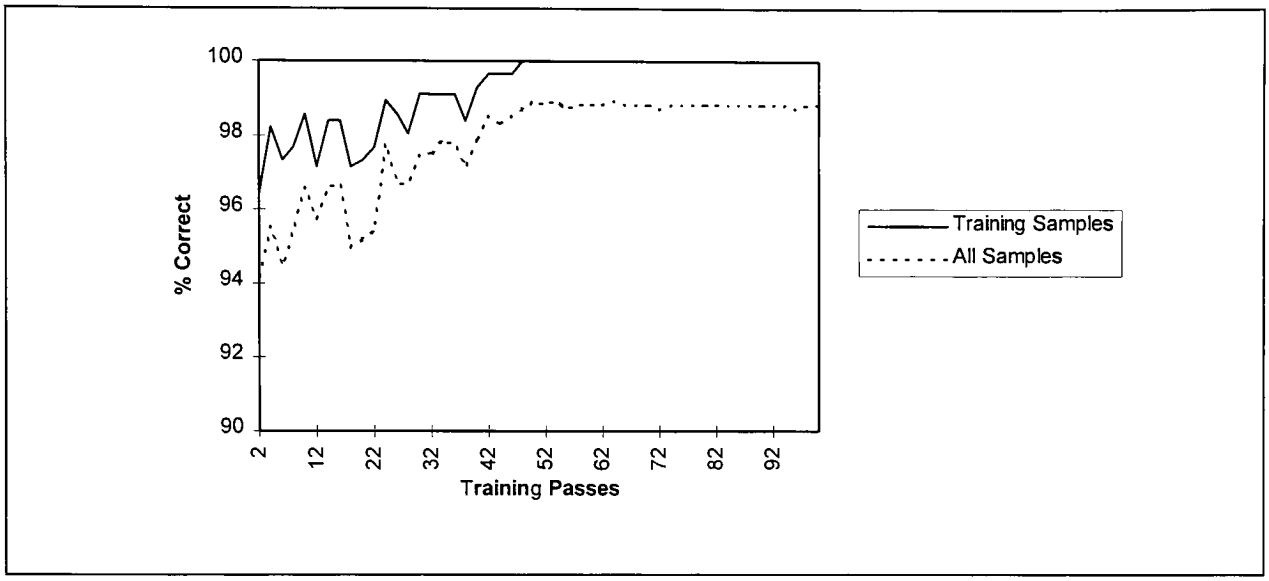
Table 20: Analysis Of Variance For Total CPU Time From Table 17

SOURCE	|------ SUMS -------|				SUM OF SQUARES	D.F.	MEAN SQUARE	M.S. Ratio
Momentum:	156	156	156	156	0.00	3	0.00	1.88
Learn Rate:	156	156	156	156	0.00	3	0.00	4.02
Nodes:	72	129	183	241	3936.77	3	1312.26	**3662115.59**
Residual:					0.00	6	0.00	
Total:				627	3936.78	15		

Figure 32 summarizes the accuracy of the classifier produced by run 12 of Table 17 during the training process. By the second pass through the training data, the network was already classifying 96.4% of the training data and 94.1%

of all samples correctly. After 48 passes through the training data, the classifier correctly classifies 100% of all training samples.

Figure 32: Classifier Accuracy By Training Pass For Best Neural Network Run Against F_1



The 16 neural network experiments conducted for feature set F_1 required a total of 627 seconds of CPU time. As with the other experiments described in this paper, this time does not include time to read in the sample data and report the results. On average, the best-of-run network was found after 71 out of the 100 passes through the training data. As mentioned earlier, the average best-of-run classifier correctly classifies 98.57% of the chest x-ray samples.

For feature set F_2, each of the best-of-run networks correctly classified 912 or 913 out of the complete set of 924 samples. Three of the 16 runs tied for best overall network with only ten incorrect responses and one response which was considered ambiguous. Of these, the network from run number two contained the minimum tested number of hidden nodes (five), and would therefore be somewhat more efficient to evaluate if used in production.

Table 21: Results Of Feedforward Neural Network Experiments Against Feature Set F_2

					BEST-OF-RUN			
RUN	MOMENT	LEARN	NODES	TOT-CPU	PASS	%TRAIN	%TEST	%TOTAL	CPU
1	0.50	0.10	15	45	32	99.28	97.84	98.70	14
2	**0.60**	**0.10**	**5**	**18**	**94**	**99.46**	**97.84**	**98.81**	**16**
3	0.70	0.10	20	60	24	99.28	97.84	98.70	14
4	0.80	0.10	10	31	98	99.46	97.84	98.81	31
5	0.50	0.20	10	31	96	99.46	97.84	98.81	30
6	0.60	0.20	15	45	86	99.46	97.84	98.81	39
7	0.70	0.20	5	18	8	99.28	97.84	98.70	1
8	0.80	0.20	20	60	16	99.28	97.84	98.70	10
9	0.50	0.35	5	18	8	99.28	97.84	98.70	1
10	0.60	0.35	20	60	16	99.28	97.84	98.70	10
11	0.70	0.35	10	31	16	99.28	97.84	98.70	5
12	0.80	0.35	15	45	40	99.28	97.84	98.70	17
13	0.50	0.50	20	60	16	99.28	97.84	98.70	10
14	0.60	0.50	10	31	84	99.46	97.84	98.81	26
15	0.70	0.50	15	45	44	99.28	97.84	98.70	20
16	0.80	0.50	5	18	86	99.28	97.84	98.70	15
				---	--	---	---	---	---
			MEAN	38	47	99.33	97.84	98.74	16

The weights for the five hidden layer and four output nodes of the best-of-run network for run two are listed in Appendix B. That experiment required only 18 seconds of CPU time. Although this network classifies fewer of the training samples correctly than the overall best neural network for feature set F_1, it correctly classifies slightly more of the samples which were not used for training (again, it must be noted that there is a slight difference in the number of unique samples for each feature set).

Table 22 summarizes the analysis of variance for the observed accuracy of best-of-run networks for feature set F_2. In this case, both the momentum and the number of nodes in the hidden layer have a statistically significant affect on the variance of the results. But, as with the neural network experiments performed for the other feature set, the variance itself is almost immeasurable.

Table 22: Analysis Of Variance For "% Total Correct" From Table 21

SOURCE	SUMS	SUM OF SQUARES	D.F.	MEAN SQUARE	M.S. Ratio
Momentum:	394.9 395.1 394.8 394.9	0.01	3	0.00	**6.33**
Learn Rate:	395.0 395.0 394.9 394.8	0.01	3	0.00	3.67
Nodes:	394.9 395.1 394.9 394.8	0.01	3	0.00	**6.33**
Residual:		0.00	6	0.00	
Total:	1579.8	0.04	15		

The results for the analyses of variance in CPU time and rate of convergence on a good solution are consistent with those for feature set F_1. The CPU time is highly dependent on the number of nodes in the hidden layer and the rate at which the simulator arrives at a good classifier is not found to depend on any one of the three parameters (within the ranges of values tested) with statistical significance. These results are summarized in the following two tables.

Table 23: Analysis Of Variance For Total CPU Time From Table 21

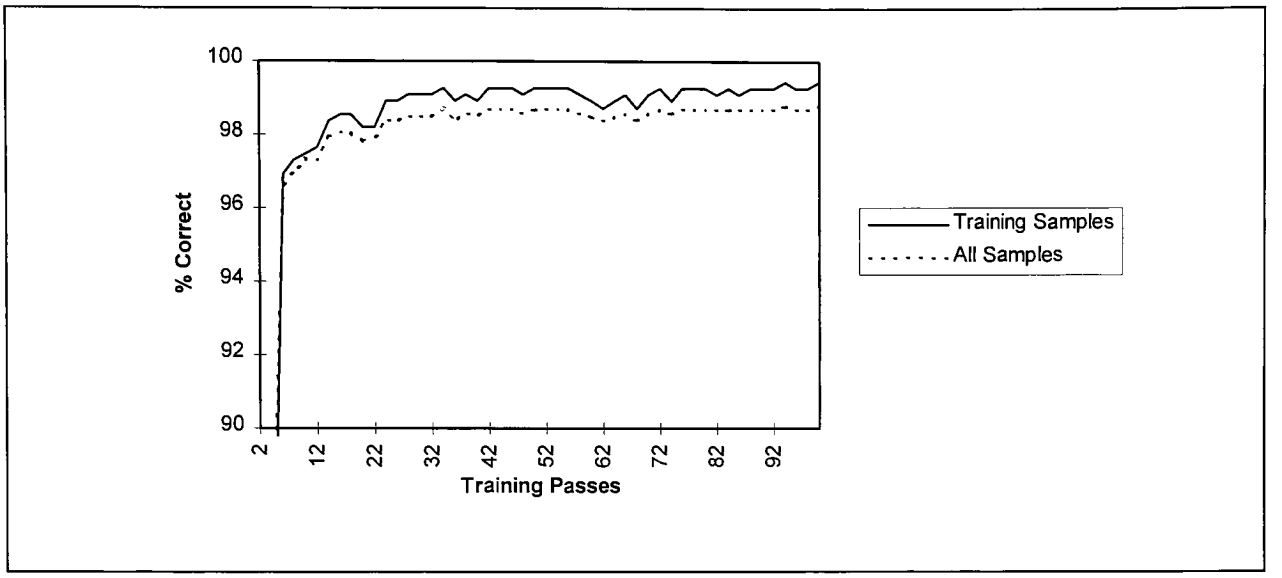
SOURCE	|------ SUMS -------|				SUM OF SQUARES	D.F.	MEAN SQUARE	M.S. Ratio
Momentum:	155	155	155	155	0.00	3	0.00	0.06
Learn Rate:	155	155	155	155	0.00	3	0.00	0.49
Nodes:	72	125	183	241	4000.56	3	1333.52	**769339.06**
Residual:					0.01	6	0.00	
Total:				623	4000.58	15		

Table 24: Analysis Of Variance For Best-Of-Run Training Pass Number From Table 21

SOURCE	|------ SUMS -------|				SUM OF SQUARES	D.F.	MEAN SQUARE	M.S. Ratio
Momentum:	152	280	92	240	5411.00	3	1803.67	3.24
Learn Rate:	248	206	230	80	4329.00	3	1443.00	2.59
Nodes:	196	294	202	72	6229.00	3	2076.33	3.73
Residual:					3342.00	6	557.00	
Total:				764	19311.00	15		

Figure 33 summarizes run number two from Table 21. In this case, the classifier required six passes through the training data to achieve a 96% rate of accuracy on the training data. This is slightly worse than was the case for the run summarized in Figure 33 and might be accounted for by smaller values for the learning rate and momentum parameters. These differences also likely explain the differences in smoothness of the curves in the two graphs.

Figure 33: Classifier Accuracy By Training Pass For Best Neural Network Run Against F_2



The 16 runs of the neural network simulator for feature set F_2 required a total of 623 seconds of CPU time. The average run resulted in a classifier which correctly classifies 98.74% of all samples. On average, the best-of-run classifier was produced after 47 out of 100 passes through the training data. This is significantly better than the corresponding result for feature set F_1. In two of the experiments, the network achieved an accuracy of 98.7% in just 8 training passes and 1.5 seconds of CPU time.

The observation that the rate of convergence on a solution does not appear to exhibit a statistically significant dependence on the learning rate or momentum factor (within the ranges that these were varied) is inconsistent with observations by Hebbar and Subhadra, who conclude:

> "The training time decreases exponentially as the learning coefficient increases. The training time is further reduced with increase in momentum factor. But after a certain increase in learning coefficient, the training algorithm becomes unstable" (Hebbar, 1992:422)

It is likely that this dependency is hidden in the results of this author's experiments by interactions between the two parameters, as was suggested earlier.

5.3. Probabilistic Neural Network Experiments

As with the previous two classification techniques, the author designed a Latin Square for evaluating the performance of the probabilistic neural network (PNN). The collection of class exemplars stored in the PNN are

:efined using learning vector quantization (LVQ), and the learning rate parameter which appears in Figure 34 is actually a parameter of this training technique (see Equation 11.2).

Figure 34: Latin Square Used In Probabilistic Neural Network Experiments

learning rate \ smoothing factor	5.0	2.0	12.0	20.0
0.10	20	10	30	5
0.20	10	30	5	20
0.50	30	5	20	10
0.30	5	20	10	30

stored exemplars per class

During some early experiments, the author found that the classifier converged much more quickly than the 500 training iterations suggested by Chettri and Cromp. For the experiments described in this section the network was initialized with randomly selected exemplars from the training data (which consisted of the first 60% of the available sample data), and the classifier was trained for just 50 passes through the training set. The remaining 40% of the samples were used to measure the ability of the resulting classifier to generalize.

The results for the 16 PNN experiments against feature set F_1 are summarized in Table 25. The overall best PNN was found during run number 14. This PNN correctly classifies only 93.8% of the samples, which is considerably worse than the classifiers found using the Genetic Programming and artificial neural network techniques. Although this classifier successfully classifies 97.33% of the training samples, its results against the test data are rather poor (only 88.53% correct). The exemplar values stored in the PNN for run 14 are listed in Appendix C.

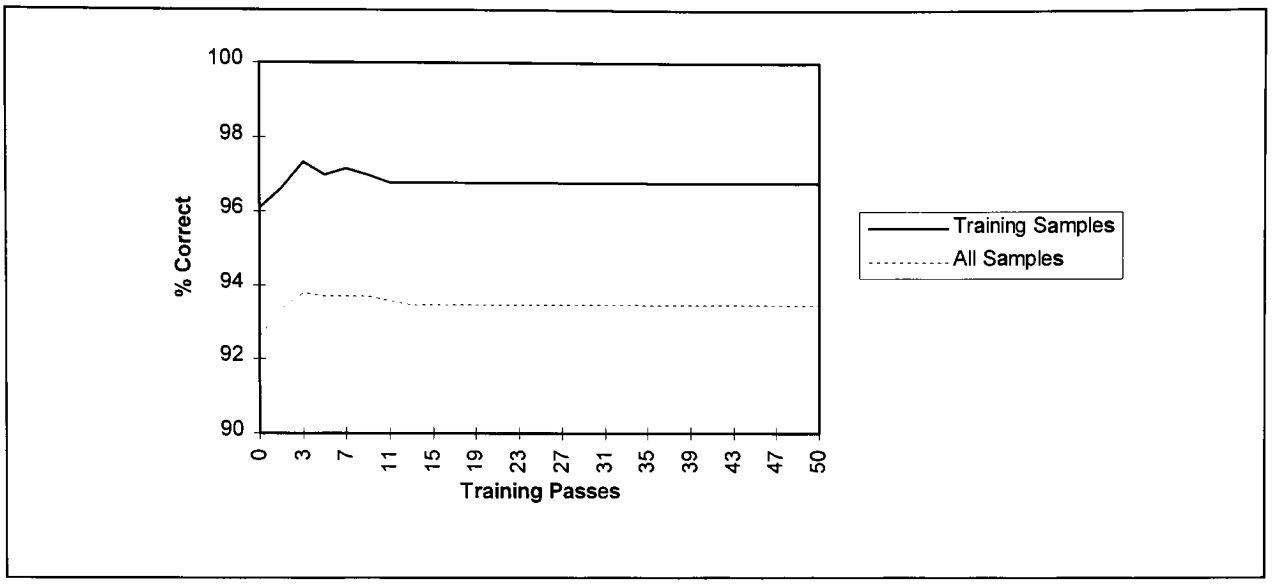
Table 25: Results Of Standard PNN Experiments Against Feature Set F_1

					BEST-OF-RUN				
RUN	SMOOTH	LEARN	EXEMPLARS	TOT-CPU	PASS	%TRAIN	%TEST	%TOTAL	CPU
1	5.0	0.10	20	42.5	1	97.15	87.47	93.27	1.2
2	2.0	0.10	10	20.7	1	97.33	88.00	93.59	0.6
3	12.0	0.10	30	65.4	5	97.15	87.20	93.16	7.0
4	20.0	0.10	5	10.5	1	96.79	88.00	93.27	0.3
5	5.0	0.20	10	20.3	9	97.15	87.73	93.38	3.8
6	2.0	0.20	30	66.4	15	97.68	87.73	93.70	20.2
7	12.0	0.20	5	10.3	1	96.79	88.80	93.59	0.3
8	20.0	0.20	20	42.6	1	96.97	87.20	93.06	1.2
9	5.0	0.50	30	65.3	9	97.68	87.47	93.59	12.1
10	2.0	0.50	5	10.8	1	95.72	89.60	93.27	0.3
11	12.0	0.50	20	42.3	1	96.43	88.00	93.06	1.2
12	20.0	0.50	10	20.4	1	96.61	88.27	93.27	0.6
13	5.0	0.30	5	10.6	1	95.54	89.60	93.16	0.3
14	**2.0**	**0.30**	**20**	**43.3**	**3**	**97.33**	**88.53**	**93.80**	**3.0**
15	12.0	0.30	10	20.4	17	97.15	87.47	93.27	7.0
16	20.0	0.30	30	65.4	5	97.50	87.47	93.48	7.0
				---	---	---	---	---	---
			MEAN	34.8	4	96.94	88.03	93.37	4.1

The CPU times in Table 25 are in units of seconds. Thus for the experiments described in this paper, the execution cost a PNN experiment is roughly the same as an artificial neural network experiment.

Surprisingly the sixth column of Table 25 suggests that the classifier tends to settle on a set of exemplars very quickly. Indeed, in the experiments conducted by the author against his two feature sets, the PNN classifier almost never showed a significant improvement in accuracy after about five training iterations. Figure 35 shows the accuracy by training iteration for the run which produced the overall best classifier in Table 25.

Figure 35: Classifier Accuracy By Training Pass For Best PNN Run Against F_1



Although one might expect the number of exemplars stored in the PNN to significantly affect the overall accuracy of the resulting classifier, this does not appear to be the case for values in the range 5 to 30 (exemplars per class). In fact, none of the three parameters by itself appears to be a significant source of variance in the ranges tested. The residual variance shown in Table 26 is likely a by-product of interactions between these parameters.

Table 26: Analysis Of Variance For "% Total Correct" From Table 25

SOURCE	|------- SUMS --------|		SUM OF SQUARES	D.F.	MEAN SQUARE	M.S. Ratio
Smoothing:	374.4 373.4 373.1 373.1		0.28	3	0.09	1.49
Learn Rate:	373.3 373.7 373.7 373.2		0.06	3	0.02	0.32
Exemplars:	373.3 373.5 373.2 373.9		0.08	3	0.03	0.44
Residual:			0.37	6	0.06	
Total:		1493.9	0.79	15		

As mentioned earlier, very little improvement in the accuracy of the classifier was observed after the first few training passes. Nevertheless, the statistic for training pass number in which the maximum accuracy is achieved does exhibit some variability in Table 25. There seems to be some dependency here on the number of exemplars per class, but the mean square ratio in Table 27 is not sufficiently large to conclude this with a reasonable degree of confidence.

Table 27: Analysis Of Variance For Best-Of-Run Training Pass Number From Table 25

SOURCE	|------- SUMS -----|				SUM OF SQUARES	D.F.	MEAN SQUARE	M.S. Ratio
Smoothing:	20	20	24	8	36.00	3	12.00	0.50
Learn Rate:	8	26	26	12	66.00	3	22.00	0.92
Exemplars:	4	28	6	34	174.00	3	58.00	2.42
Residual:					144.00	6	24.00	
Total:				72	420.00	15		

For completeness, Table 28 presents the analysis of variance for the total CPU time per run in Table 25. In this analysis, the CPU time is measured in milliseconds. The correlation between CPU time and number of exemplars in the PNN is expected. In addition, the value of the smoothing parameter appears to have a statistically significant affect on the CPU time used.

Table 28: Analysis Of Variance For Total CPU Time From Table 25

SOURCE	|--------- SUMS ----------|				SUM OF SQUARES	D.F.	MEAN SQUARE	M.S. Ratio
Smoothing:	141230	138730	138530	138910	1196200.00	3	398733.33	**11.45**
Learn Rate:	139180	139670	139700	138850	125950.00	3	41983.33	1.21
Exemplars:	42280	81830	170660	262630	7227352450.00	3	2409117483.33	**69194.37**
Residual:					208900.00	6	34816.67	
Total:				557400	7228883500.00	15		

The apparent dependency of CPU time on the value of the smoothing parameter is difficult to explain, since the smoothing parameter is only used during the evaluation of the PNN and the PNN is evaluated the same number of times in each run. Because this result is consistent with the results which will be reported for feature set F_2, it is difficult to dismiss it as a statistical aberration.

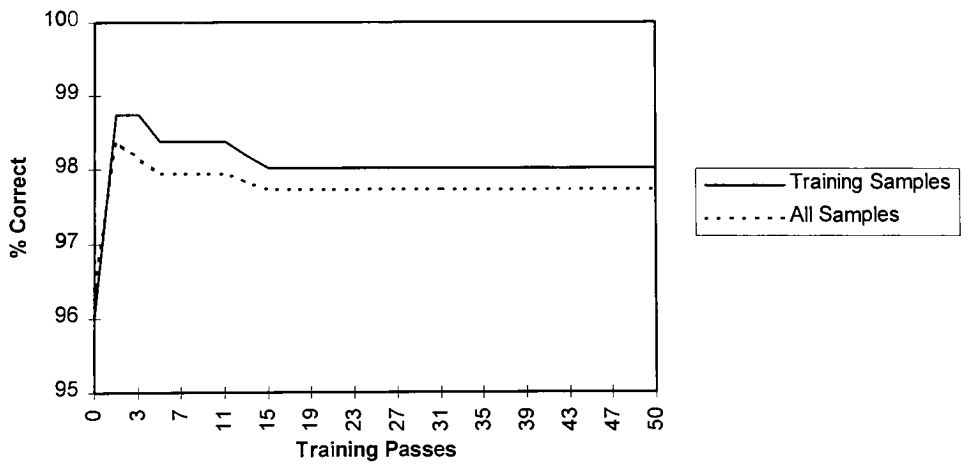
Table 29 summarizes the results of the 16 runs of the PNN simulator for feature set F_2. The overall best PNN classifier for the second feature set correctly classifies 98.38% of the sample data, including 97.84% of the samples not used for training. This is significantly better than the overall best PNN classifier for feature set F_1, and is comparable to the results achieved using Genetic Programming for feature set F_2.

Table 29: Results Of Standard PNN Experiments Against Feature Set F_2

RUN	SMOOTH	LEARN	EXEMPLARS	TOT-CPU	BEST-OF-RUN PASS	%TRAIN	%TEST	%TOTAL	CPU
1	5.0	0.10	20	40.6	1	98.56	97.30	98.05	1.2
2	2.0	0.10	10	20.4	3	98.01	96.76	97.51	1.4
3	12.0	0.10	30	64.1	1	98.01	96.76	97.51	1.8
4	20.0	0.10	5	10.5	7	98.56	97.30	98.05	1.5
5	5.0	0.20	10	20.1	9	98.01	97.03	97.62	3.8
6	2.0	0.20	30	66.1	7	98.01	97.30	97.73	9.6
7	12.0	0.20	5	10.2	5	98.38	97.84	98.16	1.1
8	20.0	0.20	20	40.9	3	98.38	97.84	98.16	2.8
9	5.0	0.50	30	64.4	1	98.56	97.57	98.16	1.9
10	2.0	0.50	5	10.7	27	94.22	96.49	95.13	5.8
11	12.0	0.50	20	40.6	1	98.38	97.57	98.05	1.2
12	20.0	0.50	10	19.8	5	85.92	96.49	90.15	2.1
13	5.0	0.30	5	10.4	19	98.38	97.57	98.05	4.0
14	**2.0**	**0.30**	**20**	**42.0**	**1**	**98.74**	**97.84**	**98.38**	**1.2**
15	12.0	0.30	10	19.9	39	95.85	97.03	96.32	15.4
16	20.0	0.30	30	64.3	1	98.38	97.30	97.94	1.9
			MEAN	34.1	8	97.15	97.25	97.19	3.5

Once again, run 14 produced the overall best classifier. It is not clear whether this is a coincidence or a result of one or a combination of the parameter values which characterize that run. Although the average experiment in Table 29 achieved its best-of-run accuracy after 8 training iterations, the classifier in run number 14 was most accurate after just a single pass through the training data. Figure 36 summarizes the behavior of this run by training pass.

Figure 36: Classifier Accuracy By Training Pass For Best PNN Run Against F_2



Both of the graphs in Figure 35 and Figure 36 show a peak in performance which is greater than the performance into which the classifier eventually settles. This might be partly explained by the fact that the training by LVQ is not directly tied to the PNN classification engine. Still, the advantage of performing at least one iteration of LVQ is clearly visible in both graphs.

The analysis of variance for the accuracy of the best-of-run classifiers (see Table 30) suggests that none of the three parameters, by itself, had a significant affect on the results of the experiments. Unlike the corresponding analysis for feature set F_1, in which the largest mean square ratio was observed for the affects of the smoothing parameter, the smoothing parameter exhibits the smallest mean square ratio in this case.

Table 30: Analysis Of Variance For "% Total Correct" From Table 29

SOURCE	SUMS				SUM OF SQUARES	D.F.	MEAN SQUARE	M.S. Ratio
Smoothing:	388.7	391.9	390.0	384.3	7.81	3	2.60	0.81
Learn Rate:	391.1	391.7	390.7	381.5	17.65	3	5.88	1.84
Exemplars:	389.4	381.6	392.6	391.3	18.34	3	6.11	1.91
Residual:					19.20	6	3.20	
Total:				1555.0	62.99	15		

Although each of the parameters individually has some effect on the rate at which the PNN simulator converges on a solution, the analysis of variance in Table 31 suggests that these individual effects are not statistically significant.

Table 31: Analysis Of Variance For Best-Of-Run Training Pass Number From Table 29

SOURCE	SUMS				SUM OF SQUARES	D.F.	MEAN SQUARE	M.S. Ratio
Smoothing:	38	30	46	16	122.75	3	40.92	0.32
Learn Rate:	12	24	60	34	312.75	3	104.25	0.81
Exemplars:	58	56	6	10	602.75	3	200.92	1.57
Residual:					769.50	6	128.25	
Total:				130	1807.75	15		

As mentioned earlier, the results for the analysis of variance of CPU time shows the same unexplained sensitivity to the value of the smoothing parameter that was observed in the experiments involving feature set F_1. However, in both cases the effect of the smoothing parameter on CPU time is negligible compared to the effect of the number of exemplars.

Table 32: Analysis Of Variance For Total CPU Time From Table 29

SOURCE	|--------- SUMS ----------|				SUM OF SQUARES	D.F.	MEAN SQUARE	M.S. Ratio
Smoothing:	139250	135570	134820	135510	3012318.75	3	1004106.25	**9.45**
Learn Rate:	135620	137410	136580	135540	587468.75	3	195822.92	1.84
Exemplars:	41850	80200	164210	258890	6968822018.75	3	2322940672.92	**21865.11**
Residual:					637437.50	6	106239.58	
Total:				545150	6973059243.75	15		

5.4. Linear Classifier Experiments

Unlike the other three classification techniques, the linear classifier is not characterized by a set of parameters from which a Latin Square can be designed. Instead, the author's linear classifier program was run a number of times, varying the fraction of sample data which was used to construct the classifier.

Of course, only the results obtained when 60% of the sample data was used in constructing the classifier can be compared with the runs of the other classifiers, but the other runs of the program offer insight into discriminating power of the two feature sets designed by the author. Table 33 summarizes the results of the runs of the linear classifier against feature set F_1.

Table 33: Results Of Linear Classifier Experiments Against Feature Set F_1

Percent Samples Used For Training	CPU (msecs)	|------ Accuracy -----| %Train	%Test	%Total	|--- # Incorrect ---| Train	Test	Total
10	120	100.0	98.1	98.3	0	16	16
20	160	98.9	96.5	97.0	2	26	28
30	210	98.6	96.3	97.0	4	24	28
40	270	98.9	95.7	97.0	4	24	28
50	330	99.1	94.0	96.6	4	28	32
60	**430**	**99.3**	**93.6**	**97.0**	**4**	**24**	**28**
70	490	99.4	91.5	97.0	4	24	28
80	550	99.5	91.5	97.9	4	16	20
90	610	97.6	95.7	97.4	20	4	24

Two things are immediately evident from the results in Table 33: the performance of the linear classifier for the chest x-ray orientation problem is comparable to that of the other techniques examined in this paper and the cost of performing a linear classifier experiment is quite negligible.

For the 60% training sample case, the linear classifier correctly classified 97% of all samples, including 93.6% of samples not used in constructing the classifier. While this is not quite as good as the results for Genetic Programming and artificial neural networks, it comes at a cost in CPU time which is about $1/100^{th}$ the cost of running the artificial neural network simulator and $1/5000^{th}$ the cost of running the Genetic Programming simulator.

The mean total accuracy of the nine runs shown above is 97.24% and the standard deviation is 0.53% (that's equivalent to just five of the 936 sample images). Clearly, the features extracted from the x-ray images are sufficiently discriminating that only a small fraction of the samples are needed to construct a reasonably good classifier. In fact, the best classifier was obtained when using only the first 10% of samples to construct the classifier.

This result suggests that the inclusion of certain samples amongst the training data can degrade the accuracy of the resulting classifier. Certainly a sample which is atypical can falsely skew the decision boundaries implicitly generated by the classification matrix.

Table 34: Results Of Linear Classifier Experiments Against Feature Set F_2

Percent Samples Used For Training	CPU (msecs)	Accuracy %Train	Accuracy %Test	Accuracy %Total	# Incorrect Train	# Incorrect Test	# Incorrect Total
10	110	100.0	96.6	97.0	0	28	28
20	160	100.0	96.2	97.0	0	28	28
30	200	100.0	95.8	97.1	0	27	27
40	280	100.0	95.7	97.4	0	24	24
50	330	99.8	96.5	98.2	1	16	17
60	**410**	**98.6**	**98.9**	**98.7**	**8**	**4**	**12**
70	470	99.4	97.1	98.7	4	8	12
80	540	99.5	97.3	99.0	4	5	9
90	610	98.6	100.0	98.7	12	0	12

Table 34 summarizes the performance of the linear classifier when used with feature set F_2. The overall accuracy in the 60% case is only slightly better than the same result for the first feature set. However, the classifier for feature set F_2 does a much better job classifying those samples which were not used in the construction of the matrix.

The classification matrices for the 60% cases in Table 33 and Table 34 are given in Appendix D.

6. A Traditional Program For Chest X-Ray Orientation

In addition to the four classification techniques described thus far, the author also implemented a traditional program for classifying the x-rays using feature set F_2. The design of the program reflects the rationale which was applied during the construction of this feature set.

As Figure 20 suggests, one expects in a correctly oriented x-ray that the column based features will all assume high values and that the row based features will assume low values near the top of the image and higher values near the bottom of the image. The first 10 features of the feature set are those that were extracted from the columns of the sample x-ray. The remaining 10 features were extracted from rows of the image.

The program includes a routine named, evalFeatures( ) which is passed a pointer to an array which contains the 20 feature values associated with a single x-ray sample. This routine classifies the orientation of the sample as to the angle through which it needs to be rotated to get it into an upright position. The behavior is as follows (the discussion assumes that features are numbered starting at one, even though the program stores the values in an array with a zero-based index):

1. Compute the sum of the first 10 features and the sum of the remaining 10 features

2. If the sum of the first 10 features is greater than the sum of the other 10, conclude that the image is either oriented correctly or is rotate 180° and proceed to step 3. Otherwise, conclude that the image must be rotated 90° or 270° and proceed to step 4.

3. Find M_1, the minimum value from amongst features 12, 13, 14, and 15. Also find the minimum value, M_2, from amongst features 17, 18, 19, and 20. If M_1 is less than M_2, conclude that the image is oriented correctly. Otherwise, if M_2 is less than M_1, conclude that the image must be rotated 180°. If M_1 and M_2 are equal, compute the sum of features 12, 13, 14, and 15 and the sum of features 17, 18, 19, and 20. Conclude that the image is oriented correctly, if and only if the first sum is less than the second. Otherwise, conclude that the image must be rotated 180°.

4. Find M_1, the minimum value from amongst features 2, 3, 4, and 5. Also find the minimum value, M_2, from amongst features 7, 8, 9, and 10. If M_1 is less than M_2, conclude that the image must be rotated 90° clockwise. Otherwise, if M_2 is less than M_1, conclude that the image must be rotated by

90° counterclockwise. If M_1 and M_2 are equal, compute the sum of features 2, 3, 4, and 5 and the sum of features 7, 8, 9, and 10. Conclude that the image must be rotated 90 degrees clockwise, if and only if the first sum is less than the second. Otherwise, conclude that the image must be rotated 90° counterclockwise.

There is a noticeable symmetry to the algorithm which is consistent with the symmetry of the feature values themselves. When run against the sample data, the program displays the following output:

Figure 37: Output Of Author's Traditional Program For X-Ray Orientation

```
          0    1    2    3
  0 :   229    0    2    0
  1 :     0  229    0    2
  2 :     2    0  229    0
  3 :     0    2    0  229
Right Answers:  916 ( 99.1 percent )
Wrong Answers:    8 (  0.9 percent )

Time Evaluate Samples: 605ms (360ms CPU time)
```

The program correctly classifies all but 8 of the 924 unique samples of feature set F_2. The classification of all 924 samples required just 360 msecs of CPU time. One could argue that the time required to design, implement, and debug the program must be considered as training cost for the classifier, in which case the author admits (without knowing the exact figures for these times) that this technique was certainly more costly than a single run of any of the other four techniques. This idea will be addressed again in section 7.3.

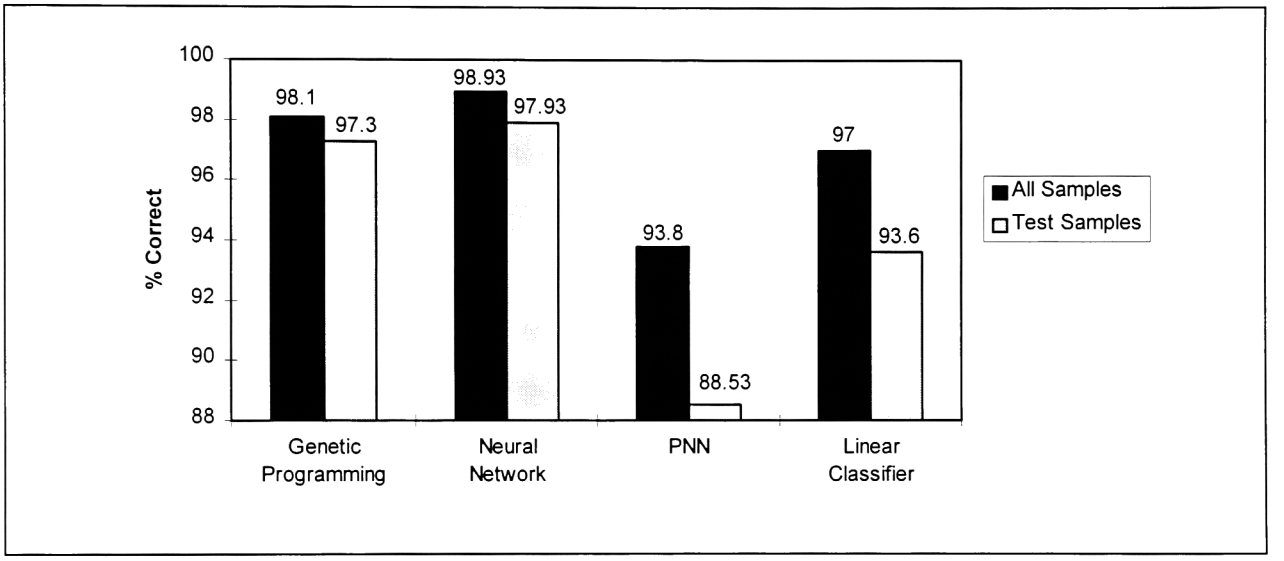
7. Conclusions

This section contains conclusions drawn from the experiments described in sections 5 and 6.

7.1. Accuracy Of The Classifiers

In the end, all of the pattern classification techniques did a reasonably good job of determining the orientation of the sample x-rays. Figure 38 and Figure 39 show the overall best results for each of the classification techniques.

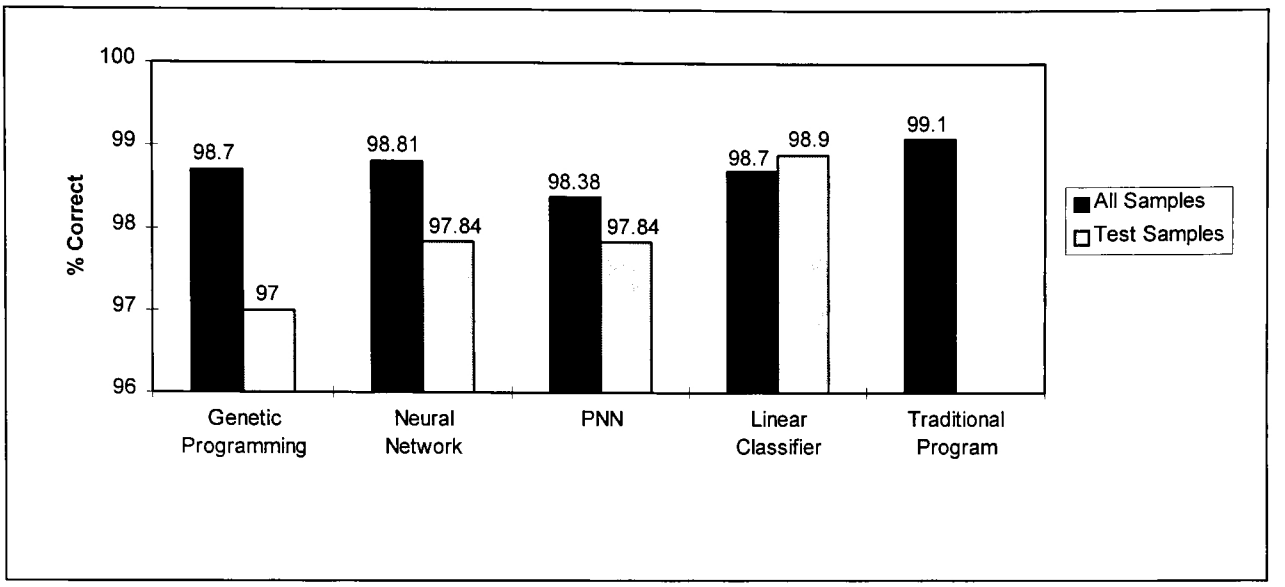
Figure 38: Overall Best Results For Feature Set F_1



For feature set F_1, the Genetic Programming and artificial neural network techniques performed better than the PNN and linear classifiers. Both of these latter two techniques resulted in classifiers which did not perform well, when tested against the samples which were not used in constructing the classifier. Scoring only 88.5% correct against these test samples, the PNN classifier is worth little when compared to the first two classifiers above.

Because more thought went into the design of feature set F_2, the author expected all four of the classification techniques to perform better when tested against that feature set. In fact, this was not the case for the overall best artificial neural networks, which correctly classify 926 out of the 936 unique samples of feature set F_1 and only 913 out of 924 unique samples of feature set F_2. Indeed, the best overall neural network for feature set F_1 is more accurate than the either of the F_1 or F_2 classifiers produced by the other three standard classification techniques.

Figure 39: Overall Best Results For Feature Set F_2



The PNN technique did much better when tested against feature set F_2, and both the PNN and linear classifiers did better at classifying the samples which were not used for training. This suggests that the samples are better clustered in the sample space defined by feature set F_2 than in the sample space defined by feature set F_1.

Of all the chest x-ray orientation classifiers which resulted from the work described in this paper, the overall best classifier was the traditional C++ program developed by the author. This program correctly classified 916 out of the 924 unique samples of feature set F_2.

7.2. Validity Of The Sample Data

All of the samples which were classified incorrectly by the author's traditional C++ program numbered among those which were used as training exemplars for the other techniques. It is possible that these samples were not good representatives of the population, in which case the author's program had an advantage over the standard classification techniques, because the construction of the program did not depend on these samples. There are actually three concerns which might be expressed over the sample data:

1. Were the 238 samples provided to the author sufficiently representative of chest x-rays in general?

2. Was rotating each image to produce four distinct samples a valid way to generate additional samples?

3. Were there images among the 238 samples which should have been filtered out of the training data, because they were atypical of the actual population of chest x-rays?

The first two questions can best be answered by obtaining a much larger set of samples and testing the author's classifiers against these samples, without rotating the samples from their original orientations. By not rotating the samples, any bias introduced by the way samples were generated for these experiments would be eliminated.

The rotating of each sample to produce additional samples, also hid from the classifiers available data concerning the a priori probability of the four different orientations. In fact, most of the sample images which were provided to the author were either right-side-up or required a 90° counterclockwise rotation to be made right-side-up. These a priori probabilities could have been used directly in the PNN classifier and would likely have improved the accuracy of the classifiers produced by the other techniques.

This latter observation is based on the fact that the majority of misclassifications were ones in which the suggested orientation was off from the true orientation by 180 degrees. If in fact the vast majority of samples fall into two classes which differ by only 90 degrees, it follows that the classifiers which have little trouble dividing the samples amongst the two super classes defined by even and odd multiples of 90 degrees would exhibit improved overall accuracy, given sample data consistent with these a priori probabilities.

Even if one assumes that the samples provided to the author were fairly representative of chest x-rays in general, there is still the possibility that, given the relatively small number of samples used in the experiments, a few bad samples could have skewed the results. The following table lists each of the samples from the original 238 for which one or more of the classifiers incorrectly classified the sample in at least one of its orientations.

The horizontal dotted line in Table 35 corresponds to the boundary between samples used for training and samples which were only used to test the classifiers. Each "x" or "o" in the table is placed at the intersection of a sample and a classifier which failed to correctly classify that sample in at least one of the sample's four orientations. The "x" is used for a sample with which at least four of the nine classifiers seemed to have a problem.

Table 35: Sample Images And The Classifiers Which Failed To Classify Them

	F1				F2					
Sample	GP	NN	PNN	LC	GP	NN	PNN	LC	PRG	Sample
5			o	o						5
14	o									14
21			o							21
83	o									83
85			o							85
104			o							104
114	o								o	114
115	o							o	o	115
121							o		o	121
124			**x**			**x**	**x**	**x**	**x**	**124**
140					o		o			140
154	o	o								154
155					o					155
161					o					161
172			o							172
173			o							173
174			o							174
182	o		o	o						182
185					o					185
186			o							186
190			o							190
192						o	o			192
197		**x**	**x**	**x**	**x**					**197**
201			o							201
204			o	o						204
207					**x**	**x**	**x**	**x**		**207**
210	**x**	**x**	**x**	**x**						**210**
218			o	o						218
219	o	o								219
223	**x**	**x**	**x**	**x**	**x**					**223**
231	o				o					231
237					o					237

There are only 32 (out of 238) sample images represented in the table and together these account for all of the misclassifications by all nine of the "overall best" classifiers. Five of the samples were targets of misclassification by at least four of the classifiers. Clearly, the features extracted from these x-rays were not sufficient to distinguish between the different orientations. Whether this indicates a deficiency in the expressive power of the feature sets or that these samples are simply not good representatives of their population is difficult to tell without further study.

Only one of these five, questionable samples was included in the data used for training. It is not clear that this one sample would be enough to disrupt the resulting classifiers. However, three of these samples are included in the set

of five samples which completely account for all errors made by the overall best classifier for feature set F_1. If these truly represent atypical samples of x-ray images and they are removed from consideration, then the corresponding neural network would correctly classify 921 out of the (now) 924 unique samples (99.7%).

7.3. The Cost Of Developing A Classifier

With regard to the cost in CPU time of training a classifier, the Genetic Programming technique was by far the most expensive of the four standard classification techniques. While each run of the Genetic Programming simulator required thousands of CPU seconds, the costs of the other techniques were measured in tens of seconds, or even hundreds of milliseconds, as was the case for the linear classifier.

If the cost of training the classifier is important, then it is difficult to justify using the Genetic Programming technique for traditional pattern classification unless it is known a priori that the other techniques will not work. However, in an absolute sense, an hour of CPU time is really not very much, and there may be measurable advantages to using the Genetic Programming technique over the other three techniques which were examined. For example, a classifier built from the program shown in Figure 26 requires only 8 out of the 20 inputs defined by feature set F_1. By reducing the feature set to these eight, the time to extract the features from a sample image is significantly reduced. Further, the program consists of only 9 simple integer operations, one of which is repeated twice. When compiled into machine language, the resulting classifier will doubtlessly require less CPU time to classify a sample than any of the other classifiers examined here.

Finally, when the result of a Genetic Programming experiment is such a small program, it may be possible to study the program and learn how the classifier actually works. This knowledge, which is otherwise hidden in neural network weights, PNN exemplars, or 20-dimensional hyper-plane decision boundaries, might provide insight into ways in which the classifier can be improved through a refinement of the feature data.

Because the C++ programs developed for the four standard classification techniques can be reused for other classification problems fairly easily, the cost of developing these programs need not be charged wholly against the problem of solving the chest x-ray orientation problem. The same cannot be said for the traditional C++ program which the author described in section 6. Although this program performed quite well, one must consider the cost of its development as part of the "training cost" of the classifier. Unfortunately, this makes the technique of traditional programming relatively expensive for solving pattern classification problems.

7.4. Sensitivity To Tunable Parameters

Sections 5.1, 5.2, and 5.3 provide detailed analyses of variance for the overall accuracy of the classifier, the rate at which the classification technique converges on a solution, and the CPU time required to execute a single experiment for the Genetic Programming, artificial neural network, and PNN techniques. This section compares the techniques based on their sensitivity to the tunable parameters which were examined in this paper.

Because it is difficult to justify the choices for the values of these parameters, a technique which is less sensitive to changes to these parameters is likely to be favored over a technique which is more sensitive to the values of the parameters which characterize it, *all else being equal.* Of course, "all else" is not equal, since the different techniques resulted in classifiers of varying degrees of accuracy. Thus, the comparison of these techniques must consider the average performance of each technique, in addition to the variance of the results. This analysis is summarized in Table 36, for the three techniques which were studied using Latin Squares.

Table 36: Best-Of-Run "% Total Correct" For Different Pattern Classification Techniques

	F_1			F_2		
Technique	Average	StdDev	Variance	Average	StdDev	Variance
GP	90.37	9.90	98.01	94.91	4.72	22.28
NN	98.57	0.27	0.07	98.74	0.05	0.00*
PNN	93.37	0.23	0.05	97.19	2.05	4.20

For the experiments described in this paper, the feedforward neural network (trained by back propagation) exhibited a better average performance, with lower variance, than did the Genetic Programming and probabilistic neural network techniques. Clearly, if only one experiment using a particular technique is to be performed, then the artificial neural network is the best choice of technique. When using the other techniques, the need for a larger suite of experiments is suggested by the relatively high variance amongst experimental results.

7.5. Ideas For Further Research

The results for the Genetic Programming experiments suggest that only a subset of the features may be needed to correctly classify a large percentage of samples. These results might be used to reduce the number of inputs to the other classification techniques. In addition, although using the raw pixel data as input to a Genetic Programming experiment may be impractical (see section 3), a feature set consisting of the raw pixel data could be used as input to a neural network or linear classifier.

The probabilistic neural network is characterized by a discriminant function which is different than that defined for the Gaussian Maximum Likelihood Classifier (GMLC) (see section 2.3.1). The author's PNN simulator could be modified to produce a GMLC and this could be used to classify x-ray orientations.

As described earlier in the paper, two results observed during these experiments are not consistent with what was described in the literature. The first concerns the rate of convergence of the feedforward neural network as a function of learning rate and momentum factor, where the author observed much less sensitivity than was suggested by Hebbar and Subhadra (Hebbar, 1992:422). It may be that some characteristic of the chest x-ray orientation problem makes it less sensitive to these parameters. It is also possible that Hebbar and Subhadra evaluated these parameters over a much wider ranges of values than was done here (their description was somewhat ambiguous).

The second inconsistency concerned the number of LVQ training iterations required for convergence of the PNN. Chettri and Cromp suggest that about 500 iterations might be required (Chettri, 1993:193). Yet in the experiments described here, improvement was rarely observed after the first few training iterations. One explanation may be that value for the number of exemplars per class needs to be much greater than the values used in these experiments (Chettri and Cromp used values in the range 4 to 80, compared to 5 through 20 here). Increasing the number of exemplars might also significantly increase the overall accuracy of the resulting classifier.

Finally, a more thorough study of the sensitivity of the different pattern classification techniques to the parameters which characterize them might be undertaken. A larger number of experiments might be defined to study the same parameters over a broader range of values or to study a larger set of these parameters. Perhaps a different technique of experimental design which does not hide information about interactions between parameters might be employed.

Appendix A: Analysis Of Classifier Produced By Genetic Programming

One of the reasons for selecting the Genetic Programming technique for a pattern classification experiment is that the resulting classifier can be analyzed to gain a better understanding of the problem. The best-of-run program which resulted from the Genetic Programming experiments against feature set F_1 consists of a parse tree of only 21 nodes, which references only eight of the 20 feature values extracted from an x-ray sample. This appendix considers how this program manages to correctly classify the orientation of over 98% of the sample chest x-rays.

The program was previously described via a LISP-like s-expression in Figure 26. This expression is repeated immediately below:

```
(& (+ (| (| 1 (> i2 i8)) (> i15 i11)) (> i15 i11)) (| (> i20 i17) (> i19 i14)))
```

The "+" function is simply integer addition, and the "&" and "|" functions represent bitwise AND and OR operators respectively. The ">" function returns -1 if the first argument is greater than the second. Otherwise, it returns zero. The constant "1" appears in one place in the program.

Feature values extracted from the x-ray sample are labeled with numbers which are prefixed by the letter "`i`". Inputs `i1` through `i10` represent counts of the number white pixels along different columns of the cropped and thresholded image. Inputs `i11` through `i20` represent counts of the number of white pixels along rows of that image. The program above references only two of the column based features and six of the row based features.

The x-ray sample is assigned a particular orientation based on the value of the low-order two bits of the program result, as follows: 00 if the sample is oriented correctly, 01 if the sample needs to be rotated 90° counterclockwise, 10 if the sample needs to be rotated 180°, and 11 if the sample needs to be rotated 270° counterclockwise.

Ultimately, the program returns the bitwise AND of these two sub-expressions:

```
(+ (| (| 1 (> i2 i8)) (> i15 i11)) (> i15 i11))
```

and

```
(| (> i20 i17) (> i19 i14))
```

This quickly leads to three observations about the behavior of the program:

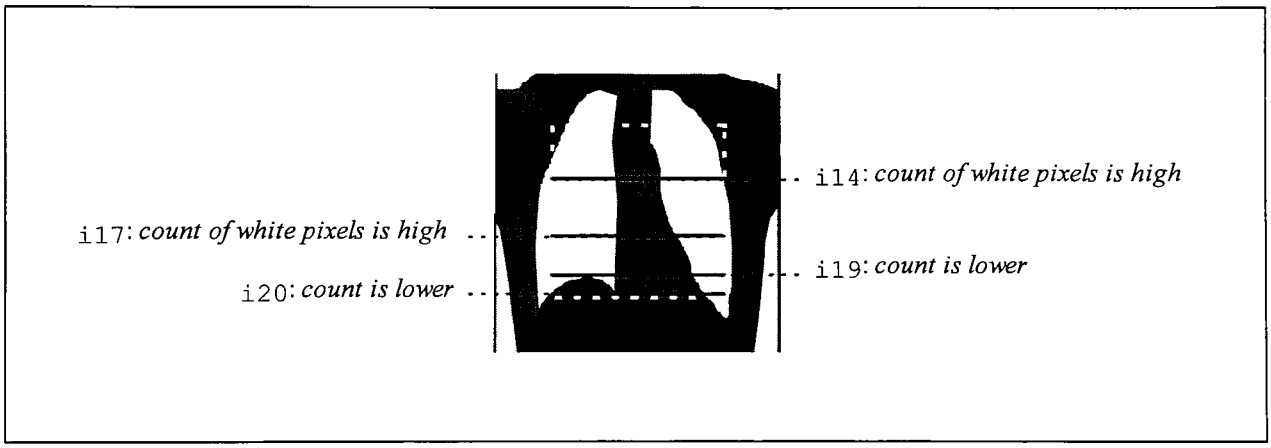
1. In any case where `i20` is less than or equal to `i17` *and* `i19` is less than or equal to `i14`, the program returns the value zero (because the second sub-expression is zero).

2. In any case where `i20` is greater than `i17` *or* `i19` is greater `i14`, the value of the program is equal to the value of the first sub-expression (because the second sub-expression consists entirely of 1-bits).

3. In any case where `i15` is greater than `i11`, the value of the first sub-expression is -2 ($11...10_2$).

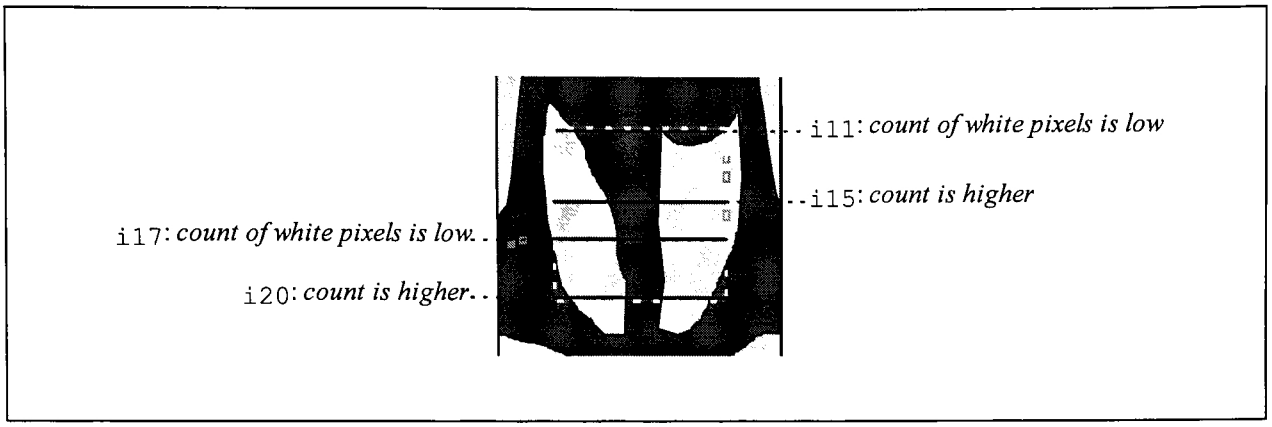
The goal of performing the threshold operation on the sample x-ray image was to produce a sample image in which the lungs appear white and the rest of the central portion of the image appears black. Row based features of the correctly oriented sample are expected to be high near the top of the image and lower near the bottom, where the lungs are occluded by other organs. If this hypothesis is correct, one would expect feature `i20` to be less than feature `i17` and feature `i19` to be less than feature `i14`. Thus for a correctly oriented image, the program would produce a result of zero, as per the first observation above.

Figure 40: Example Of Correctly Oriented X-Ray



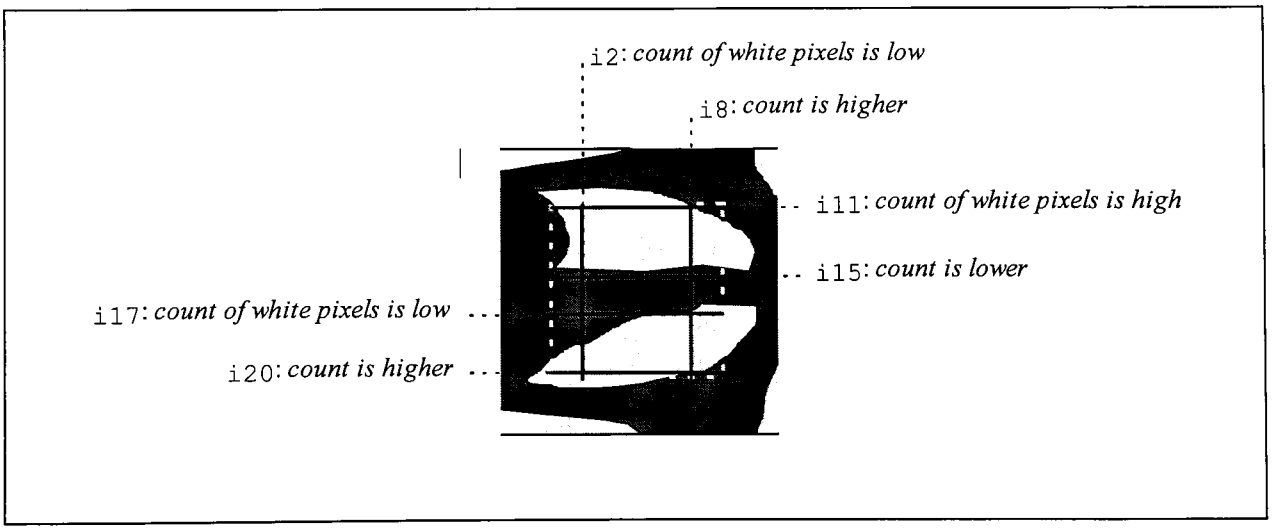
The magnitudes of the row based features are exactly reversed when the orientation of the sample is off by 180°. In this case, we can expect the second and third observations about the program to apply. That is, it is likely that `i20` is greater than `i17` or `i19` is greater than `i14`, in which case the program result is equal to the value of the first sub-expression. Because it is also likely that `i15` will be greater than `i11`, the value of the first sub-expression is expected to be -2. The two low order bits of -2 in twos-complement representation are 10, so the classifier correctly classifies these samples.

Figure 41: Example Of A X-Ray Whose Orientation Is Off By 180°



When the orientation of the x-ray is off by 90° or 270° counterclockwise, it is likely that the row corresponding to feature `i15` will cross the central dark area between the lungs, resulting in a low feature value. Similarly, one might expect at least one of the two rows which correspond to features `i14` and `i17` to cross near this dark region, while features `i11`, `i19`, and `i20` are likely to cross the lungs longitudinally, resulting in high feature values.

Figure 42: Example Of X-Ray Whose Orientation If Off By 90°

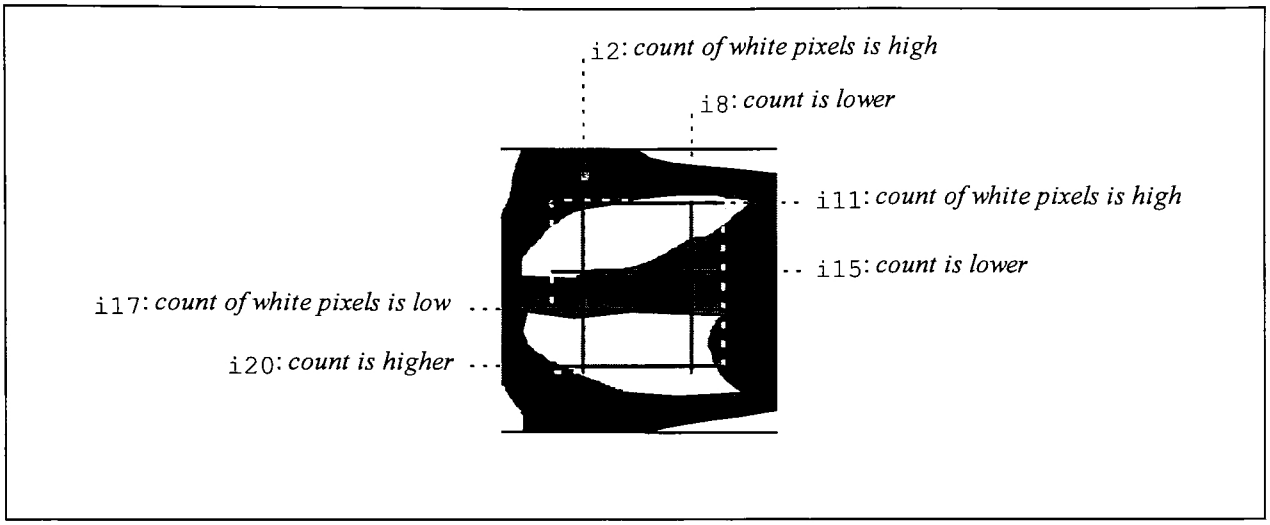


From the second observation on the program's behavior, one concludes that the result of the program is likely to be equivalent to the value of the first sub-expression. Since it is also likely that `i15` is less than `i11`, this sub-expression reduces to:

```
(| 1 (> i2 i8))
```

For the 90° case (see Figure 42) one expects the feature value `i2` to be less than `i8`, in which case the result of evaluating the program is 1. For the 270° case (Figure 43) one expects `i2` to be greater than `i8`, in which case the result of the program is -1 (whose two low order bits are 11).

Figure 43: Example Of X-Ray Whose Orientation Is Off By 270°



Thus in all cases, the program appears to discriminate the orientation of the samples in a manner which is consistent with what was described as the rationale for the design of feature set F_1 in section 3.2.

Appendix B: Neural Network Weights

This appendix lists the network weights associated with the overall best neural networks for feature sets F_1 and F_2. The first 20 weights listed for each hidden layer node (reading left to right and then top-down) represent the weights applied to the 20 feature values of the x-ray sample. The 21st weight is the weight applied to the bias input. Each output layer node defines one weight for the output of each hidden layer node and one weight for its bias input.

Hidden Layer For Feature Set F_1:

node 1:
1.0749 0.6742 -1.3473 -0.4259 0.8606 -2.6059 -1.2573 0.0520 -0.8534 0.8260
-0.2492 0.5115 0.1722 -0.8604 0.2777 2.2824 1.9099 0.8622 -1.3349 -4.3067 0.3806

node 2:
1.4713 -0.4341 1.0578 -1.2409 -2.6723 0.5251 -1.7195 -3.7644 1.0360 2.3495
-7.8356 -2.6330 -0.1703 3.6551 3.1592 -0.8402 -2.4075 1.3584 2.3561 2.1498 -1.1777

node 3:
3.4289 3.3138 2.6387 -0.1641 0.7498 0.9016 0.1643 -2.1568 -3.2298 -5.7436
-1.5719 -1.2411 -0.0190 -1.1387 -0.3561 0.8632 -0.3205 -2.1174 1.8975 1.6768 0.2592

node 4:
-0.2864 -0.2057 -0.3692 -0.5870 -0.1562 0.4241 -0.4187 -0.4396 -0.3948 -0.3871
0.0492 -0.8639 -0.7259 -0.4242 -0.5034 0.1157 -0.1410 0.4584 0.1724 -0.0338 -0.6495

node 5:
1.3450 -0.2219 1.8699 -2.6556 -2.7810 1.6092 2.7043 -1.3912 0.5514 -1.2059
-3.6128 -3.6094 -1.8253 -0.7070 -0.5313 0.3994 1.2626 1.2039 2.9033 3.3933 -0.1205

node 6:
2.2060 -0.3731 1.6546 -2.7903 -4.4322 0.6445 0.9677 -2.0651 0.6941 0.8913
-6.5026 -2.3742 0.1021 3.2038 3.4481 -0.1956 -2.0360 0.5019 0.8550 1.5390 -0.3207

node 7:
-2.9751 -0.1154 0.2645 1.0023 2.5225 3.9821 2.5111 0.4696 -1.3559 -5.8395
4.9245 1.3442 0.4157 -3.0387 -5.0758 -2.4221 -0.3757 -1.8748 0.1749 3.8983 0.1813

node 8:
-1.1068 -0.2303 0.1455 0.4370 0.1775 -0.2125 -0.2309 -0.6321 0.7062 0.3102
-0.1420 -0.7047 -0.7395 0.4999 0.1156 -1.4360 -0.5654 0.9708 0.0277 0.5356 -0.1821

node 9:
-3.0801 -0.1514 -2.3391 3.6206 2.5528 -5.0870 -5.3432 2.0313 2.0273 4.6863
1.1542 1.3979 -0.8640 -0.6278 2.3532 1.5325 1.4222 0.1235 -2.9426 -4.7455 0.7050

node 10:
-5.5031 -3.4088 -1.4565 1.2916 1.9975 0.8581 -0.1052 0.9386 2.3097 2.8651
0.4279 0.6000 -1.8167 1.3814 1.6263 -2.7860 -2.4360 1.9366 -0.6693 0.9506 -0.4650

node 11:
-4.0424 -2.8045 -0.2360 0.7528 1.8838 0.9243 0.4737 0.8091 1.3146 1.0037
1.1917 0.7629 -1.7688 2.1380 0.3473 -2.8834 -3.3487 1.2522 0.2766 1.1619 -0.1577

node 12:
-3.3141 -0.7427 -0.1175 2.2316 3.6940 1.0028 -0.7036 0.2527 -0.4451 -1.4067
2.9257 0.6190 -1.7444 -2.6488 -3.7826 -1.0697 0.1348 1.3429 1.3910 2.2609 -0.6714

node 13:
3.7899 2.7861 2.4264 0.2346 1.5864 1.4012 0.3332 -2.8463 -5.0641 -7.7341
0.1600 -1.0215 1.5982 -2.1656 -2.1450 1.0518 -0.2816 -3.0825 1.9771 1.5675 0.3593

Hidden Layer For Feature Set F_1 (continued):

node 14:
1.8683 1.4591 -2.6653 -0.8940 1.4669 -2.0867 -1.7499 0.9531 -1.5649 0.7620
3.1022 2.5963 1.7721 -0.1884 0.6762 1.2382 1.1538 -1.5003 -4.2105 -7.0129 1.0470

node 15:
5.4842 1.6324 -0.8490 -2.6690 -3.4494 -2.0117 -0.6090 0.7047 -0.2006 2.1757
-0.4569 0.3430 2.2907 -2.3330 -1.0846 4.7766 5.5489 -0.3526 -2.1673 -4.7092 -0.4813

Output Layer For Feature Set F_1:

node 1 (0°):
1.9612 -4.4181 -2.0423 -0.6407 -2.9376 -1.8264 -3.8269 -1.2342 3.4776 -1.9490
-1.9566 0.7884 -0.3237 1.9115 3.6805 -2.9958

node 2 (90° counterclockwise):
-2.6427 -0.0689 -3.9397 -0.1780 -0.1315 -3.0166 0.4677 0.6444 1.3904 2.0750
1.7431 1.9854 -4.9937 -1.2856 -5.9384 0.0550

node 3 (180°):
-0.7953 4.0449 0.7970 -0.4721 1.8455 2.4688 -3.2999 -0.4559 -3.5314 0.0044
-0.1037 -5.0155 -0.7192 -4.3640 0.4402 0.1911

node 4 (270° counterclockwise):
-2.8642 -3.7479 1.7192 0.3395 -0.5603 -3.6396 4.5185 -1.0492 -4.2468 -3.0963
-2.7270 0.6964 3.0106 -0.5178 1.9915 -3.7148

Hidden Layer For Feature Set F_2:

node 1:
3.9957 1.2115 1.4567 2.2088 1.2555 -1.2007 -2.3826 -1.4252 -2.0832 -3.2513
5.4131 2.5194 2.4496 2.5256 0.8844 -1.1617 -2.4213 -2.4829 -3.0098 -5.2516 0.3897

node 2:
0.3981 0.6076 0.6056 1.2735 1.0390 1.1237 1.2025 0.9223 0.0934 0.2613
-0.7547 -0.9326 -1.0945 -0.8650 -0.9306 -0.5714 -0.9424 -0.5729 -0.2236 -0.6333 0.0129

node 3:
-0.3900 -0.6299 -0.4625 -1.0642 -0.9917 -0.5876 -0.9261 -0.8789 -0.9679 -0.3085
0.8277 0.9070 0.8969 1.0887 0.7583 1.3744 0.7305 0.2251 0.5155 0.1242 -0.0741

node 4:
-4.5724 -4.0397 -3.0670 -2.1599 -1.1101 0.7824 2.4668 3.3928 3.9687 4.3241
1.0186 -0.2842 0.2278 0.9780 -0.1940 0.0557 -0.3916 -0.1763 0.1712 -1.2102 -0.0029

node 5:
2.1926 1.2772 0.9814 0.7033 -0.1578 -0.0344 -0.5898 -1.0345 -1.8404 -1.4578
-3.2606 -2.7761 -2.4772 -1.6793 -1.2286 1.2547 2.4840 3.1827 3.5725 1.1121 -0.2410

Output Layer For Feature Set F_2:

node 1 (0°):
-6.3681 3.8554 -4.5099 -3.6028 4.6847 -1.2231

node 2 (90° counterclockwise):
4.0508 -5.0445 3.0914 -7.0501 0.9039 -1.8920

node 3 (180°):
4.9476 3.1492 -4.6936 1.2188 -6.9387 -2.7464

node 4 (270° counterclockwise):
-5.5570 -5.1822 2.7826 5.4537 -2.5276 -1.4153

Appendix C: Probabilistic Neural Network Exemplars

This appendix lists the exemplars stored in the overall best PNN classifiers for feature sets F_1 and F_2. In both cases, there were 20 exemplars per class. An unclassified sample is first normalized according to Equation 17 and then the values of the four discriminant functions are computed using Equation 9, where the value of the smoothing parameter is 2.0. The results of the four discriminant functions are compared and the sample is assigned to the class associated with the largest of these results.

Because of the large amount of data in this appendix, the exemplars associated with each class are presented on their own page. Each exemplar is a 1x20 vector which was found using the LVQ technique. These results correspond to the runs numbered 14 in Table 25 (feature set F_1) and Table 29 (feature set F_2).

PNN Exemplars For Feature Set F_1:

Class 1 (0°):

<0.9257, 0.9101, 0.9071, 0.8293, 0.3347, 0.1075, 0.1812, 0.2786, 0.6349, 0.8394,
0.9623, 0.8594, 0.7437, 0.6824, 0.6349, 0.6178, 0.5397, 0.5020, 0.4390, 0.1744>

<0.9429, 0.9264, 0.9089, 0.7543, 0.2426, 0.1844, 0.2629, 0.4671, 0.6892, 0.8087,
0.9760, 0.9254, 0.8194, 0.7102, 0.6867, 0.6327, 0.5573, 0.4892, 0.4057, 0.1646>

<0.9837, 0.9524, 0.9524, 0.9314, 0.2279, 0.0241, 0.0484, 0.1854, 0.5403, 0.7687,
0.7131, 0.7687, 0.7706, 0.6102, 0.6390, 0.6020, 0.5630, 0.4667, 0.4005, 0.3896>

<0.8095, 0.8095, 0.8095, 0.7822, 0.2108, 0.0882, 0.1632, 0.5855, 0.9321, 0.9321,
0.9727, 0.8298, 0.7143, 0.6870, 0.6667, 0.6190, 0.6190, 0.5987, 0.2654, 0.1093>

<1.0000, 0.9277, 0.5853, 0.1479, 0.0575, 0.1223, 0.7120, 1.0000, 0.9759, 1.0000,
0.8021, 0.8490, 0.8045, 0.6895, 0.7186, 0.6852, 0.6519, 0.5845, 0.5506, 0.4659>

<0.9997, 0.8208, 0.6154, 0.4159, 0.3863, 0.3939, 0.5681, 0.9562, 0.9579, 0.9759,
0.9990, 0.9973, 0.9953, 0.9430, 0.7232, 0.6546, 0.5686, 0.5340, 0.4530, 0.4403>

<0.8994, 0.9932, 0.9932, 0.9932, 0.9731, 0.0200, 0.0000, 0.0000, 0.0001, 0.2667,
0.5262, 0.5695, 0.5940, 0.5153, 0.5306, 0.5151, 0.4987, 0.4762, 0.4762, 0.4562>

<1.0000, 1.0000, 1.0000, 0.5714, 0.2381, 0.0952, 0.0952, 0.4762, 0.7619, 0.8095,
0.6667, 0.9048, 0.8571, 0.7619, 0.6667, 0.6190, 0.6190, 0.4762, 0.3810, 0.3810>

<1.0000, 1.0000, 1.0000, 0.7578, 0.3441, 0.2418, 0.3022, 0.6366, 0.8803, 0.9310,
0.9945, 0.9910, 0.8892, 0.7861, 0.7367, 0.6908, 0.6432, 0.5742, 0.5359, 0.3983>

<0.8554, 0.8097, 0.7808, 0.6174, 0.4579, 0.3272, 0.3347, 0.4404, 0.5197, 0.6015,
1.0000, 0.9890, 0.9833, 0.8530, 0.7474, 0.4824, 0.4044, 0.2384, 0.2094,-0.0605>

<0.9805, 0.9553, 0.8277, 0.5112, 0.2077, 0.2310, 0.4811, 0.7021, 0.7823, 0.8694,
0.9682, 0.9432, 0.9101, 0.8291, 0.7460, 0.6771, 0.6137, 0.5193, 0.3635, 0.2112>

<0.8134, 0.7922, 0.7788, 0.7760, 0.3339, 0.1795, 0.1999, 0.3328, 0.6994, 0.7945,
0.9951, 0.9577, 0.7664, 0.7046, 0.6472, 0.6095, 0.5341, 0.4388, 0.1692, 0.0012>

<0.7367, 0.5122, 0.1037, 0.0527, 0.1698, 0.4268, 0.8073, 0.8302, 0.8773, 0.8784,
0.9328, 0.7446, 0.6952, 0.6712, 0.6202, 0.5972, 0.6202, 0.4683, 0.2330, 0.0224>

<0.9425, 0.9173, 0.8574, 0.7621, 0.5913, 0.1951, 0.0713, 0.2184, 0.4102, 0.5195,
0.9789, 0.8923, 0.7484, 0.6925, 0.5782, 0.5070, 0.4386, 0.4005, 0.2759, 0.1423>

<0.9843, 0.9652, 0.9646, 0.8613, 0.2555, 0.1497, 0.1942, 0.4129, 0.7639, 0.9527,
0.8863, 0.9163, 0.8632, 0.6867, 0.6850, 0.6647, 0.6163, 0.5570, 0.4982, 0.3964>

<0.7795, 0.7475, 0.7475, 0.7561, 0.3122, 0.0281, 0.0033, 0.1411, 0.4611, 0.7600,
0.8141, 0.7197, 0.6767, 0.6378, 0.6036, 0.5640, 0.5208, 0.4025, 0.0000, 0.0000>

<0.9768, 0.9674, 0.9674, 0.9668, 0.5933, 0.0117, 0.0104, 0.1720, 0.3753, 0.6128,
0.7962, 0.7956, 0.7262, 0.6493, 0.6041, 0.5757, 0.4999, 0.4471, 0.4046, 0.3914>

<0.7702, 0.6059, 0.4516, 0.3302, 0.3438, 0.4660, 0.6036, 0.6969, 0.7965, 0.8831,
1.0000, 0.9895, 0.9630, 0.8501, 0.7458, 0.5963, 0.4478, 0.2941, 0.1741, 0.0618>

<0.9880, 0.9871, 0.9801, 0.8694, 0.1774, 0.1308, 0.3489, 0.9933, 0.9988, 1.0000,
0.8977, 0.8142, 0.7530, 0.7043, 0.6710, 0.6521, 0.6676, 0.6859, 0.7265, 0.8920>

<0.9796, 0.9637, 0.9604, 0.7211, 0.3835, 0.3006, 0.3424, 0.4721, 0.6704, 0.8109,
0.9807, 0.9937, 0.9764, 0.7661, 0.7121, 0.6345, 0.5644, 0.4672, 0.3944, 0.3613>

PNN Exemplars For Feature Set F_1 (continued):

Class 2 (90° counterclockwise):

<0.0807, 0.1894, 0.3004, 0.4638, 0.5911, 0.7039, 0.8070, 0.9522, 0.9919, 1.0000,
0.7724, 0.6013, 0.4233, 0.2727, 0.3345, 0.4644, 0.6072, 0.7051, 0.8242, 0.9076>

<0.0154, 0.1390, 0.2310, 0.3701, 0.5201, 0.6839, 0.8467, 0.9539, 0.9794, 1.0000,
0.8567, 0.8012, 0.7179, 0.6190, 0.4756, 0.3283, 0.3745, 0.4386, 0.4641, 0.5426>

<0.0953, 0.3657, 0.5337, 0.5794, 0.6207, 0.6538, 0.6743, 0.7431, 0.8692, 0.9542,
0.8526, 0.8489, 0.8396, 0.8138, 0.2597, 0.1004, 0.1733, 0.4258, 0.7686, 0.8963>

<0.3550, 0.3895, 0.4415, 0.5190, 0.5770, 0.6504, 0.5830, 0.7608, 0.7452, 0.7132,
0.9935, 0.9480, 0.9474, 0.9567, 0.2704, 0.0010, 0.0015, 0.0719, 0.4517, 0.7104>

<0.3892, 0.3938, 0.4322, 0.5143, 0.5868, 0.6212, 0.6268, 0.7275, 0.7770, 0.8085,
0.9631, 0.9559, 0.9559, 0.9460, 0.5543, 0.0071, 0.0000, 0.1535, 0.3835, 0.6415>

<0.4692, 0.5601, 0.6209, 0.6639, 0.7135, 0.7159, 0.6994, 0.7469, 0.8200, 0.8928,
0.9939, 0.9792, 0.7216, 0.3061, 0.1064, 0.1397, 0.6164, 0.9326, 0.9659, 0.9774>

<0.3645, 0.4295, 0.4681, 0.5567, 0.6095, 0.6336, 0.6524, 0.7686, 0.8341, 0.8229,
0.9601, 0.9511, 0.9524, 0.9287, 0.3180, 0.0381, 0.0973, 0.2711, 0.6024, 0.7985>

<0.4598, 0.4762, 0.4762, 0.4956, 0.5172, 0.5306, 0.5221, 0.5909, 0.5716, 0.5278,
0.9150, 0.9932, 0.9932, 0.9932, 0.9767, 0.0163, 0.0000, 0.0000, 0.0001, 0.2776>

<0.2642, 0.3876, 0.4855, 0.5958, 0.6436, 0.7548, 0.7971, 0.9415, 0.9835, 0.9982,
0.9733, 0.9568, 0.9294, 0.6245, 0.3130, 0.2775, 0.3873, 0.5458, 0.7184, 0.8733>

<0.0212, 0.2328, 0.4656, 0.6244, 0.5979, 0.6190, 0.6720, 0.6931, 0.7461, 0.9365,
0.7354, 0.5239, 0.1059, 0.0529, 0.1693, 0.4233, 0.8095, 0.8307, 0.8783, 0.8783>

<0.0347, 0.3333, 0.2381, 0.4744, 0.5074, 0.8389, 0.9671, 1.0000, 1.0000, 1.0000,
1.0000, 0.9212, 0.8571, 0.5879, 0.4450, 0.2710, 0.3810, 0.4597, 0.5697, 0.7272>

<0.0888, 0.2082, 0.3334, 0.5192, 0.6444, 0.8120, 0.9630, 0.9965, 1.0000, 1.0000,
0.9036, 0.7709, 0.6194, 0.4511, 0.3770, 0.4375, 0.5395, 0.6758, 0.7603, 0.8253>

<0.0000, 0.0001, 0.4146, 0.5221, 0.5572, 0.5986, 0.6395, 0.6736, 0.7161, 0.8113,
0.7842, 0.7496, 0.7496, 0.7575, 0.3142, 0.0284, 0.0000, 0.1385, 0.4494, 0.7636>

<0.4456, 0.4611, 0.5269, 0.5665, 0.6527, 0.7337, 0.9258, 1.0000, 1.0000, 1.0000,
1.0000, 0.8045, 0.5953, 0.4017, 0.3872, 0.3806, 0.5576, 0.9545, 0.9559, 0.9720>

<0.3447, 0.4087, 0.4671, 0.5450, 0.6577, 0.6864, 0.7298, 0.8957, 0.9840, 0.9829,
0.9686, 0.9530, 0.9525, 0.7907, 0.3500, 0.2167, 0.2869, 0.4125, 0.6618, 0.8187>

<0.9887, 0.7259, 0.6768, 0.6597, 0.6368, 0.6682, 0.7002, 0.7472, 0.8072, 0.9110,
1.0000, 1.0000, 1.0000, 0.9582, 0.1717, 0.1261, 0.3226, 1.0107, 0.9973, 0.9992>

<0.0000, 0.0364, 0.0454, 0.2571, 0.4195, 0.5714, 0.6982, 0.8684, 0.9405, 0.9909,
0.7184, 0.7184, 0.6639, 0.6163, 0.5056, 0.3270, 0.1905, 0.2494, 0.3558, 0.4510>

<0.2905, 0.3807, 0.5711, 0.6605, 0.6868, 0.6868, 0.8247, 0.9787, 0.9744, 1.1501,
1.0000, 0.9904, 0.9904, 0.9868, 0.3868, 0.1872, 0.3265, 0.4924, 0.7175, 0.9489>

<0.1162, 0.2770, 0.3718, 0.4390, 0.5301, 0.5871, 0.6914, 0.7635, 0.8969, 0.9778,
0.9195, 0.9062, 0.8762, 0.7794, 0.5721, 0.1992, 0.0367, 0.2074, 0.4241, 0.5407>

<0.4064, 0.4402, 0.5316, 0.5963, 0.6607, 0.6743, 0.6732, 0.8431, 0.9186, 0.8144,
0.9825, 0.9600, 0.9600, 0.8596, 0.2227, 0.0892, 0.1579, 0.4033, 0.7197, 0.9171>

PNN Exemplars For Feature Set F_1 (continued):

Class 3 (180°):

<0.7611, 0.6632, 0.4970, 0.3454, 0.2753, 0.3984, 0.7063, 0.9487, 0.9700, 0.9779,
0.3474, 0.3634, 0.4301, 0.5412, 0.6406, 0.7046, 0.7981, 0.9391, 0.9967, 0.9990>

<0.2946, 0.0000, 0.0000, 0.0000, 0.0091, 0.9794, 0.9885, 0.9885, 0.9885, 0.9073,
0.4671, 0.4762, 0.4762, 0.5122, 0.5327, 0.5353, 0.5034, 0.6074, 0.5918, 0.5288>

<0.9998, 0.9991, 0.9903, 0.3432, 0.1244, 0.1708, 0.8534, 0.9657, 0.9976, 0.9979,
0.9104, 0.7092, 0.6860, 0.6668, 0.6643, 0.6755, 0.6886, 0.7306, 0.7945, 0.8939>

<0.8845, 0.8845, 0.8368, 0.8095, 0.4356, 0.1632, 0.0406, 0.0812, 0.4684, 0.7416,
0.0273, 0.2451, 0.4902, 0.6120, 0.5917, 0.6190, 0.6597, 0.6870, 0.7276, 0.9181>

<0.8944, 0.8147, 0.6914, 0.5477, 0.4349, 0.3702, 0.4126, 0.6017, 0.7452, 0.8637,
0.1042, 0.2301, 0.3860, 0.5163, 0.6366, 0.7876, 0.9488, 0.9881, 0.9982, 1.0000>

<0.7443, 0.5068, 0.1434, 0.0048, 0.0086, 0.2756, 0.9727, 0.9524, 0.9528, 0.9767,
0.3870, 0.3974, 0.4590, 0.5443, 0.5833, 0.6232, 0.6028, 0.7238, 0.7582, 0.7398>

<0.8127, 0.5489, 0.1493, 0.0723, 0.0365, 0.3636, 0.8337, 0.8698, 0.8698, 0.8809,
0.0303, 0.3553, 0.5112, 0.5242, 0.5818, 0.6301, 0.6650, 0.7059, 0.8252, 0.8598>

<0.5961, 0.5444, 0.4586, 0.3751, 0.3300, 0.4526, 0.6299, 0.7803, 0.8354, 0.8883,
0.0663, 0.1907, 0.2433, 0.3889, 0.5444, 0.7488, 0.8837, 0.9739, 0.9948, 1.0000>

<0.8896, 0.6810, 0.3565, 0.2214, 0.1329, 0.2392, 0.8218, 0.9719, 0.9718, 0.9884,
0.3908, 0.4517, 0.5189, 0.5847, 0.6355, 0.6687, 0.6747, 0.8532, 0.9418, 0.8294>

<0.4664, 0.3474, 0.3006, 0.2554, 0.3465, 0.5114, 0.5757, 0.6503, 0.7428, 0.7634,
-0.0302, 0.0864, 0.1164, 0.2553, 0.4144, 0.5968, 0.7424, 0.8974, 0.9314, 0.9938>

<0.6905, 0.4590, 0.0008, 0.0000, 0.0281, 0.6028, 0.8812, 0.9369, 0.9369, 0.9844,
0.2642, 0.4003, 0.4444, 0.5515, 0.6147, 0.6892, 0.6582, 0.6992, 0.7394, 0.7384>

<0.9927, 0.9817, 0.9700, 0.6496, 0.1590, 0.1028, 0.2481, 0.6899, 0.9743, 1.0000,
0.5186, 0.5623, 0.6210, 0.6530, 0.7069, 0.7143, 0.6970, 0.7532, 0.8192, 0.8871>

<0.9334, 0.9315, 0.5913, 0.1613, 0.0848, 0.2097, 0.7790, 0.8086, 0.8092, 0.8094,
0.1179, 0.2664, 0.5977, 0.6182, 0.6189, 0.6661, 0.6846, 0.7151, 0.8295, 0.9705>

<0.5512, 0.4353, 0.2177, 0.0592, 0.2106, 0.5515, 0.7834, 0.8688, 0.9096, 0.9215,
0.1046, 0.2961, 0.3731, 0.4322, 0.5331, 0.5986, 0.6897, 0.7672, 0.8880, 0.9774>

<0.8997, 0.7261, 0.3807, 0.2200, 0.1672, 0.2854, 0.7905, 0.8734, 0.8875, 0.8965,
0.0592, 0.4444, 0.5364, 0.5678, 0.6376, 0.7066, 0.6724, 0.8109, 0.9474, 0.9496>

<0.9338, 0.8625, 0.6898, 0.4249, 0.2248, 0.2720, 0.6004, 0.9160, 0.9622, 0.9776,
0.3161, 0.4810, 0.5563, 0.6345, 0.6833, 0.7359, 0.8067, 0.9092, 0.9693, 0.9811>

<0.7586, 0.4920, 0.1233, 0.0000, 0.0199, 0.3057, 0.7423, 0.7292, 0.7293, 0.7655,
0.0000, 0.0001, 0.3128, 0.5270, 0.5810, 0.6122, 0.6259, 0.6617, 0.7111, 0.8064>

<0.9828, 0.9801, 0.9725, 0.5387, 0.3566, 0.3984, 0.4465, 0.6864, 0.8830, 0.9930,
0.4680, 0.4960, 0.5397, 0.5780, 0.6536, 0.7482, 0.9066, 0.9891, 0.9995, 0.9997>

<0.6185, 0.3285, 0.0919, 0.0007, 0.0051, 0.6059, 0.9575, 0.9575, 0.9575, 0.9769,
0.4104, 0.4123, 0.4342, 0.5081, 0.5819, 0.6078, 0.5945, 0.7263, 0.7343, 0.7478>

<0.5238, 0.4762, 0.4286, 0.2857, 0.1905, 0.2381, 0.9048, 0.9524, 0.9048, 0.9048,
0.1905, 0.3810, 0.3810, 0.4762, 0.4762, 0.7143, 0.6190, 0.8095, 0.8095, 0.9524>

PNN Exemplars For Feature Set F_1 (continued):

Class 4 (270° counterclockwise):

<0.8508, 1.1128, 1.0025, 0.7179, 0.6357, 0.5635, 0.6157, 0.3489, 0.1991, 0.1217,
0.7151, 0.5793, 0.3707, 0.2067, 0.1712, 0.2610, 0.5572, 0.9866, 0.9866, 0.9937>

<1.0000, 1.0000, 1.0000, 0.8957, 0.7365, 0.6505, 0.5610, 0.5167, 0.4626, 0.4616,
0.9791, 0.9747, 0.9728, 0.5369, 0.3656, 0.3967, 0.4361, 0.6421, 0.8698, 1.0000>

<0.9981, 0.9887, 0.9075, 0.7766, 0.7028, 0.6316, 0.5390, 0.4396, 0.3636, 0.3246,
0.7691, 0.6480, 0.4574, 0.3264, 0.2685, 0.3696, 0.6865, 0.9444, 0.9650, 0.9660>

<0.9765, 0.8858, 0.7735, 0.6928, 0.6036, 0.5413, 0.4379, 0.3813, 0.3312, 0.1002,
0.5903, 0.4656, 0.2400, 0.0863, 0.2105, 0.4985, 0.7981, 0.8760, 0.9056, 0.9135>

<0.7694, 0.8476, 0.7756, 0.6291, 0.6605, 0.6080, 0.5571, 0.4671, 0.4091, 0.3830,
0.7798, 0.6117, 0.2862, 0.0656, 0.0185, 0.2900, 0.9353, 0.9529, 0.9525, 0.9637>

<0.8829, 0.9633, 0.8584, 0.6952, 0.6693, 0.6516, 0.5928, 0.5325, 0.4613, 0.4283,
0.9507, 0.7289, 0.3981, 0.2400, 0.1730, 0.2238, 0.8482, 0.9812, 0.9819, 0.9836>

<1.0000, 0.9983, 0.9883, 0.9339, 0.7831, 0.6286, 0.4986, 0.3662, 0.2221, 0.0876,
0.9061, 0.8269, 0.7054, 0.5612, 0.4394, 0.3732, 0.4093, 0.5594, 0.6900, 0.8223>

<0.8932, 0.7913, 0.7317, 0.6911, 0.6709, 0.6513, 0.6668, 0.6865, 0.7224, 0.9583,
1.0000, 1.0000, 0.9957, 0.3275, 0.1233, 0.1746, 0.9051, 0.9834, 0.9990, 0.9992>

<0.7845, 0.7601, 0.7314, 0.6445, 0.6319, 0.5760, 0.5149, 0.4453, 0.3947, 0.3761,
0.6526, 0.4052, 0.1494, 0.0000, 0.0076, 0.5062, 0.9556, 0.9618, 0.9621, 0.9811>

<0.5245, 0.5904, 0.6075, 0.5085, 0.5331, 0.5334, 0.5122, 0.4762, 0.4762, 0.4677,
0.2989, 0.0002, 0.0000, 0.0000, 0.0084, 0.9820, 0.9907, 0.9907, 0.9907, 0.9115>

<0.9982, 0.9694, 0.9519, 0.8351, 0.7125, 0.4450, 0.4108, 0.2748, 0.2439 -0.0535,
0.5727, 0.4677, 0.4132, 0.3164, 0.3014, 0.4757, 0.6150, 0.7811, 0.8505, 0.8900>

<0.9845, 0.9235, 0.8429, 0.6992, 0.5714, 0.4131, 0.2443, 0.0774, 0.0619, 0.0000,
0.4457, 0.3504, 0.2238, 0.1905, 0.3313, 0.4929, 0.6056, 0.6532, 0.7461, 0.7461>

<0.9690, 0.9506, 0.9054, 0.8388, 0.7410, 0.6619, 0.5960, 0.4711, 0.3351, 0.2363,
0.8748, 0.7834, 0.6723, 0.4895, 0.2369, 0.2286, 0.4965, 0.7998, 0.9467, 0.9949>

<1.0000, 0.9952, 0.9661, 0.9029, 0.8065, 0.6128, 0.3962, 0.2046, 0.0751,-0.0073,
0.6380, 0.5962, 0.4964, 0.4121, 0.3415, 0.4478, 0.6112, 0.6919, 0.7438, 0.8242>

<0.9904, 0.8947, 0.7325, 0.6944, 0.6588, 0.6085, 0.6006, 0.5312, 0.2241, 0.0371,
0.8955, 0.8584, 0.4577, 0.1914, 0.1423, 0.2578, 0.7847, 0.7941, 0.8019, 0.8123>

<0.9440, 0.8999, 0.7739, 0.6868, 0.6682, 0.6169, 0.5589, 0.5181, 0.4381, 0.0515,
0.8797, 0.6707, 0.3238, 0.1873, 0.1285, 0.2849, 0.8096, 0.8759, 0.8913, 0.8960>

<0.9236, 0.9081, 0.8389, 0.7751, 0.7504, 0.6983, 0.6525, 0.6110, 0.5216, 0.3826,
0.8902, 0.8185, 0.5996, 0.1469, 0.1217, 0.3139, 0.9277, 0.9995, 0.9995, 0.9995>

<0.8164, 0.7212, 0.6803, 0.6278, 0.6104, 0.5690, 0.5170, 0.4141, 0.0003, 0.0001,
0.7685, 0.4780, 0.1435, 0.0000, 0.0218, 0.3075, 0.7625, 0.7497, 0.7497, 0.7774>

<0.8868, 0.8493, 0.7792, 0.7045, 0.7095, 0.7016, 0.6784, 0.6328, 0.6207, 0.4464,
0.9941, 0.9854, 0.9651, 0.6542, 0.1507, 0.1257, 0.3030, 0.7296, 0.9552, 0.9735>

<0.9357, 0.7486, 0.6963, 0.6723, 0.6208, 0.5976, 0.6221, 0.4675, 0.2359, 0.0240,
0.8803, 0.8794, 0.8293, 0.8053, 0.4247, 0.1707, 0.0541, 0.1107, 0.5231, 0.7408>

PNN Exemplars For Feature Set F_2:

```
Class 1 (0°):

    <1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000,
     0.1600, 0.1549, 0.0816, 0.1206, 0.0284, 0.0536, 0.1636, 0.2317, 0.3081, 0.4173>

    <0.0793, 0.3933, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000,
     0.1452, 0.1548, 0.1152, 0.1061, 0.0769, 0.0797, 0.2038, 0.7360, 1.0211, 1.0000>

    <1.0000, 1.0000, 1.0000, 0.0277, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000,
     0.1677, 0.1848, 0.1320, 0.1519, 0.0993, 0.1172, 0.2251, 0.2782, 0.4507, 1.0000>

    <1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000,
     0.1397, 0.1658, 0.1138, 0.0995, 0.0722, 0.0616, 0.1572, 0.9285, 0.9633, 0.9729>

    <1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000,
     0.1297, 0.1382, 0.1032, 0.1142, 0.0797, 0.1224, 0.2162, 0.5208, 0.6907, 0.8912>

    <1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000,
     0.1777, 0.1582, 0.1033, 0.1490, 0.0353, 0.0611, 0.1764, 0.2165, 0.2973, 0.3137>

    <0.2785, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000,
     0.1690, 0.1546, 0.0952, 0.1378, 0.0638, 0.0547, 0.1345, 0.2226, 0.2967, 0.7616>

    <1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000,
     0.1606, 0.1778, 0.1427, 0.1525, 0.0526, 0.1049, 0.1773, 0.2542, 0.3124, 0.3484>

    <1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000,
     0.1504, 0.1656, 0.1232, 0.1220, 0.0334, 0.0656, 0.1753, 0.2504, 0.3091, 0.6371>

    <1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000,
     0.0422, 0.1187, 0.0950, 0.1087, 0.0838, 0.1487, 0.3064, 0.4096, 0.5415, 0.5486>

    <1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000,
     1.0000, 1.0000, 0.3862,-0.1348,-0.0477, 0.2663, 0.4698, 0.7063, 1.0000, 1.0000>

    <1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000,
     0.1614, 0.1668, 0.1096, 0.1275, 0.0464, 0.0807, 0.1830, 0.2318, 0.3955, 0.9288>

    <1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000,
     0.8938, 0.0954, 0.0434, 0.1005, 0.0901, 0.1566, 0.4050, 0.5592, 0.7741, 0.8411>

    <1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000,
     0.1196, 0.1669, 0.1085, 0.1212, 0.0670, 0.1276, 0.2461, 0.3835, 0.9994, 0.8556>

    <1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000,
     0.1393, 0.1700, 0.1171, 0.1274, 0.0526, 0.0873, 0.1925, 0.2289, 0.3028, 0.4167>

    <1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 0.2223, 0.1000, 1.0000,
     0.1471, 0.2012, 0.1581, 0.1620, 0.1233, 0.1691, 0.2451, 0.3062, 0.3223, 0.1422>

    <1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000,
     0.1365, 0.1438, 0.0954, 0.1220, 0.0518, 0.0618, 0.1888, 0.2399, 0.3191, 0.3355>

    <1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000,
     0.0169, 0.0919, 0.2560, 0.0508, 0.0555, 0.1613, 0.9093, 0.9999, 1.0000, 1.0000>

    <1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 0.3500, 1.0000,
     1.0000, 0.0377, 0.0605, 0.1094, 0.1992, 0.9780, 1.0000, 1.0000, 1.0000, 1.0000>

    <1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000,
     0.1502, 0.1338, 0.0756, 0.1175, 0.0455, 0.0527, 0.1568, 0.2371, 0.3106, 0.3109>
```

PNN Exemplars For Feature Set F_2 (continued):

Class 2 (90° counterclockwise):

<0.9816, 0.9990, 0.3436, 0.2533, 0.0972, 0.0498, 0.0913, 0.0779, 0.1378, 0.6951,
1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000>

<0.3184, 0.3113, 0.2382, 0.1559, 0.0618, 0.0526, 0.1239, 0.0807, 0.1438, 0.1555,
1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000>

<0.6301, 0.2952, 0.2240, 0.1697, 0.0584, 0.0449, 0.1220, 0.1025, 0.1601, 0.1550,
1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000>

<1.0000, 0.8382, 0.3171, 0.2384, 0.1459, 0.0849, 0.1652, 0.1061, 0.1485, 0.1434,
1.0000, 1.0000, 1.0000, 0.0942, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000>

<0.3256, 0.3084, 0.2381, 0.1951, 0.0961, 0.0526, 0.1622, 0.1345, 0.1805, 0.1778,
1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000>

<0.9939, 0.7318, 0.2246, 0.2003, 0.1126, 0.0785, 0.1103, 0.1250, 0.1806, 0.1391,
1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000>

<0.4328, 0.2922, 0.2400, 0.1944, 0.0810, 0.0526, 0.1220, 0.1209, 0.1576, 0.1553,
1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000>

<0.3489, 0.3268, 0.2374, 0.1765, 0.0530, 0.0327, 0.1352, 0.1100, 0.1483, 0.1316,
1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000>

<0.3139, 0.3113, 0.2382, 0.1932, 0.0874, 0.0529, 0.1464, 0.1220, 0.1581, 0.1556,
1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000>

<1.0000, 1.0000, 1.0000, 0.1854, 0.0697, 0.0789, 0.0976, 0.1190, 0.1633, 0.1411,
0.4078, 0.3933, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000>

<1.0000, 1.0000, 1.0000, 1.0000, 0.1468, 0.0591, 0.0419, 0.6700, 0.1209, 0.0191,
1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000>

<1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 0.1866, 0.1314, 0.0806, 0.0972, 1.0000,
1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 0.9204, 1.0000>

<1.0000, 1.3568, 0.2986, 0.1995, 0.0963, 0.0691, 0.1252, 0.0622, 0.1093, 0.0911,
0.1040, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000>

<0.9999, 0.9999, 0.9997, 0.2686, 0.1396, 0.0978, 0.1310, 0.1025, 0.1258, 0.0883,
1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000>

<0.4172, 0.3182, 0.2152, 0.1335, 0.0810, 0.0263, 0.0906, 0.0899, 0.1438, 0.1438,
1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000>

<0.4438, 0.3451, 0.2335, 0.2110, 0.1164, 0.0447, 0.0885, 0.0299, 0.0833, 0.2321,
1.0000, 0.6382, 1.4333, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000>

<0.8909, 0.8711, 0.2235, 0.6470, 0.3185, 0.0317,-0.0396,-0.6013, 0.4698, 1.0000,
1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000>

<0.3820, 0.3633, 0.2710, 0.1604, 0.0533, 0.0263, 0.1187, 0.0490, 0.0866, 0.0960,
0.6959, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.5778>

<0.5862, 0.5447, 0.3701, 0.2445, 0.1271, 0.0627, 0.1126, 0.0953, 0.1402, 0.1307,
1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000>

<0.2965, 0.2891, 0.2150, 0.1707, 0.0534, 0.0285, 0.1464, 0.1190, 0.1701, 0.1556,
1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000>

PNN Exemplars For Feature Set F_2 (continued):

Class 3 (180°):

<1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000,
 0.9993, 0.9812, 0.2618, 0.2095, 0.0704, 0.0443, 0.0744, 0.0892, 0.1417, 0.1386>

<1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000,
 0.3309, 0.3087, 0.2381, 0.1789, 0.1041, 0.0526, 0.1518, 0.1396, 0.1783, 0.1612>

<1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 0.7877, 0.0946,
 0.9987, 1.0154, 0.4208, 0.1846, 0.0810, 0.0618, 0.1174, 0.0970, 0.1635, 0.1566>

<1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000,
 0.8396, 0.8076, 0.5568, 0.4011, 0.1587, 0.0901, 0.1026, 0.0484, 0.0972, 0.8958>

<1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000,
 0.4698, 0.3071, 0.2455, 0.1830, 0.0908, 0.0438, 0.1181, 0.1319, 0.1689, 0.1543>

<1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 0.2018, 1.0000, 1.0000, 1.0000,
 0.9524, 0.7517, 0.3023, 0.2298, 0.1345, 0.0789, 0.1592, 0.1071, 0.1501, 0.1479>

<1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000,
 0.2895, 0.2973, 0.2143, 0.1707, 0.0547, 0.0263, 0.1385, 0.1109, 0.1726, 0.1631>

<1.0000, 0.3500, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000,
 0.9963, 0.9957, 0.9954, 0.9923, 0.9249, 0.1766, 0.0989, 0.0524, 0.0349, 1.0000>

<1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000,
 0.3256, 0.3033, 0.2298, 0.1707, 0.0697, 0.0263, 0.1463, 0.1007, 0.1700, 0.1727>

<1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000,
 1.0000, 1.0000, 1.0000, 0.2407, 0.1167, 0.0792, 0.0986, 0.0733, 0.1237, 0.1132>

<1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000,
 0.3256, 0.3105, 0.2374, 0.1882, 0.0526, 0.0459, 0.1174, 0.0907, 0.1333, 0.1354>

<1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000,
 1.0000, 1.0000, 1.0000, 1.0000, 0.1639, 0.0933, 0.1053, 0.0974, 0.1353, 0.1202>

<1.2441, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 0.3623,
 0.3637, 0.3477, 0.2533, 0.1513, 0.0432, 0.0374, 0.1202, 0.0804, 0.1084, 0.1000>

<1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000,
 0.9797, 0.9756, 0.5151, 0.2775, 0.1083, 0.0478, 0.0802, 0.0287, 0.0595, 0.0117>

<1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000,
 0.6376, 0.3430, 0.2416, 0.2043, 0.1060, 0.0988, 0.1277, 0.1434, 0.1807, 0.1448>

<1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000,
 0.7161, 0.6991, 0.6279, 0.4731, 0.3495, 0.1188, 0.1031, 0.0788, 0.0980, 0.1011>

<1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000,
 0.9998, 0.6509, 0.2585, 0.2058, 0.1124, 0.0917, 0.1010, 0.1158, 0.1664, 0.1298>

<1.0000, 0.1716, 0.5202, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000,
 0.1820, 0.3555, 0.3345, 0.2734, 0.1716, 0.1178, 0.1479, 0.1416, 0.1755, 0.0997>

<1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000,
 0.3552, 0.3474, 0.2536, 0.1870, 0.0697, 0.0498, 0.1232, 0.1026, 0.1690, 0.1522>

<1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000,
 0.3114, 0.3111, 0.2381, 0.1896, 0.0961, 0.0526, 0.1454, 0.1194, 0.1615, 0.1606>

PNN Exemplars For Feature Set F_2 (continued):

Class 4 (270° counterclockwise):

<0.1239 0.1402 0.0953 0.1061 0.0355 0.0597 0.1736 0.2381 0.3259 0.3478,
1.0000 1.0000 1.0000 1.0000 1.0000 1.0000 1.0000 1.0000 1.0000 1.0000>

<0.0434 0.1234 0.1119 0.1481 0.1072 0.1776 0.3303 0.3807 0.5139 1.0000,
1.0000 1.0000 1.0000 1.0000 1.0000 1.0000 1.0000 1.0000 1.0000 1.5633>

<0.1462, 0.1512, 0.1061, 0.1592, 0.0785, 0.1330, 0.2294, 0.3054, 0.7538, 1.0000,
1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 0.2018, 1.0000, 1.0000, 1.0000>

<0.1701, 0.1871, 0.1218, 0.1457, 0.0466, 0.0813, 0.1867, 0.2399, 0.2916, 0.3149,
1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000>

<0.0271, 0.0974, 0.0720, 0.1195, 0.1023, 0.1903, 0.2942, 1.0000, 1.0000, 1.0000,
1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000>

<0.1384, 0.1333, 0.0714, 0.1190, 0.0323, 0.0526, 0.1652, 0.2352, 0.3084, 0.3111,
1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000>

<-0.4062,-0.3695,-0.4536, 0.0007, 0.0447, 0.0849, 0.2135, 1.0000, 1.0000, 1.0000,
1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000>

<0.1540, 0.1577, 0.1180, 0.1421, 0.0523, 0.0857, 0.1948, 0.2333, 0.2914, 0.4316,
1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000>

<0.1747, 0.0777, 0.0111, 0.0730, 0.0136, 0.1334, 0.2110, 0.2245, 0.3366, 0.4324,
1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.4333, 0.6382, 1.0000>

<0.9956, 0.0730, 0.0529, 0.0990, 0.1320, 0.9245, 0.9906, 0.9949, 0.9957, 0.9968,
1.0000, 0.7725, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000>

<0.1604, 0.1554, 0.0977, 0.1215, 0.0455, 0.0551, 0.1667, 0.2163, 0.2912, 0.5460,
1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000>

<0.0406, 0.0548, 0.0385, 0.1419, 0.1344, 0.2640, 0.3830, 0.5434, 0.9658, 0.9224,
1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000>

<0.1011, 0.1272, 0.0915, 0.1053, 0.0933, 0.1703, 0.9134, 1.0000, 1.0000, 1.0000,
1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000>

<0.1444, 0.1664, 0.1368, 0.1193, 0.0895, 0.1101, 0.2218, 0.2597, 0.4239, 1.0000,
1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000>

<0.0393, 0.1099, 0.0836, 0.1044, 0.0757, 0.1443, 0.2958, 0.3973, 0.5108, 0.5303,
1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000>

<0.0939, 0.0839, 0.0534, 0.1143, 0.0563, 0.0583, 0.1445, 0.3104, 0.4998, 0.7812,
1.5778, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 0.9571, 0.2078>

<0.0878, 0.1672, 0.1337, 0.1284, 0.1086, 0.1631, 0.3000, 0.3560, 0.3880, 0.2735,
1.0000, 0.1420, 0.9195, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000>

<0.2305, 0.0750, 0.0144, 0.0038, 0.0295, 0.0032, 0.2060, 0.2335, 0.3257, 0.4146,
1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 0.3864, 1.5200, 1.0000>

<0.8931, 0.3307, 0.0617, 0.0884, 0.0509, 0.1243, 0.2670, 0.3589, 0.9994, 1.0000,
1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000>

<0.1142, 0.1357, 0.0903, 0.0922, 0.0618, 0.1177, 0.2582, 0.3577, 0.6005, 0.7685,
1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000>

Appendix D: Linear Classifier Matrices

This appendix lists the linear classifier matrices for the 60% training cases of feature sets F_1 and F_2. Each matrix contains 21 rows and four columns. A sample is first converted into homogeneous coordinates, by appending a constant "1.0" to the end of the 1x20 feature vector. This new vector is post multiplied by the classifier matrix, resulting in a 1x4 output vector. The class associated with the column of the result vector containing the maximum response is the class to which the sample is assigned.

0°	90°	180°	270°	(counterclockwise)

Feature Set F_1:

0.0131159	-0.0628695	0.0209616	0.0287919
-0.0058911	0.0013107	-0.0182918	0.0228722
-0.0081923	-0.0053352	0.0027025	0.0108250
0.0010709	0.0271868	-0.0347414	0.0064836
0.0064797	0.0443314	-0.0379708	-0.0128402
-0.0372407	-0.0133021	0.0065298	0.0440130
-0.0351150	0.0069109	0.0005612	0.0276429
0.0057962	0.0098334	-0.0104265	-0.0052031
-0.0184098	0.0230859	-0.0062518	0.0015756
0.0220715	0.0285358	0.0120621	-0.0626693
0.0281383	0.0126434	-0.0622608	0.0214791
0.0228313	-0.0055476	0.0017092	-0.0189929
0.0094399	-0.0099385	-0.0047649	0.0052634
0.0065318	0.0017672	0.0277808	-0.0360799
-0.0137043	0.0056911	0.0449776	-0.0369644
0.0439331	-0.0367348	-0.0126833	0.0054850
0.0268304	-0.0358835	0.0074925	0.0015606
-0.0056423	0.0054782	0.0102420	-0.0100779
0.0009318	-0.0188636	0.0236966	-0.0057648
-0.0634114	0.0216880	0.0293746	0.0123489
-0.4959514	-0.4998903	-0.5064910	-0.4976673

Feature Set F_2:

-0.0020136	0.0210384	0.0009964	-0.0200212
-0.0012361	0.0053654	0.0006698	-0.0047992
0.0016592	0.0075946	-0.0006033	-0.0086505
0.0007292	0.0012965	0.0000433	-0.0020691
0.0104059	0.0236174	0.0035203	-0.0375436
0.0027799	-0.0359319	0.0108760	0.0222760
0.0005267	-0.0031329	0.0006355	0.0019707
-0.0003561	-0.0096032	0.0011308	0.0088284
0.0005964	-0.0045588	-0.0010486	0.0050110
0.0010351	-0.0199928	-0.0020163	0.0209740
-0.0199953	-0.0019906	0.0209893	0.0009966
-0.0045350	-0.0010994	0.0050111	0.0006233
-0.0096311	0.0012012	0.0088365	-0.0004066
-0.0031664	0.0007009	0.0019659	0.0004996
-0.0358775	0.0106983	0.0222300	0.0029492
0.0236588	0.0033654	-0.0375934	0.0105692
0.0013097	0.0000487	-0.0020439	0.0006855
0.0075838	-0.0005389	-0.0086197	0.0015748
0.0053654	0.0006379	-0.0048231	-0.0011802
0.0210228	0.0010336	-0.0200184	-0.0020379
-0.4960479	-0.4928518	-0.5039521	-0.5071482

Bibliography

Bailey, D. and Thompson, D. (1990) "How to develop neural network applications", AI Expert, June: 38-46.

Caudill M. (1991) "Neural network training tips and techniques", AI Expert, January: 56-61.

Chettri, S. R. and Cromp, R. F. (1993) "Probabilistic neural network architecture for high-speed classification of remotely sensed imagery", Telematics and Informatics, Vol 10, no. 3: 187-198.

Finney, D. J. (1960) An Introduction to the Theory of Experimental Design, University of Chicago Press, Chicago.

Goldberg, D. E. (1994) "Genetic and evolutionary algorithms come of age", Communications of the ACM, Vol. 37, no. 3: 113-119.

Gonzalez, R. C. and Wintz, P. (1987) Digital Image Processing - 2nd Edition, Addison-Wesley, Reading.

Hebbar, J. K. and Subhadra P. (1992) "A study of the optimum use of various parameters of back propagation ANN and improvements to training time for multispectral classification of remote sensing satellite data", Proceedings of the 2nd Singapore International Conference on Image Processing: 421-425.

Holland, J. H. (1992) Adaptation In Natural And Artificial Systems, The MIT Press, Cambridge.

Jain, A. K. (1989) Fundamentals of Digital Image Processing, Prentice-Hall, Englewood Cliffs.

Johnson, N. L. and Leone, F. C. (1977) Statistical and Experimental Design in Engineering and the Physical Sciences Volume II (2nd Edition), John Wiley & Sons, New York.

Keith, M. J. and Martin, M. C. (1994) "Genetic programming in C++: implementation issues", Advances In Genetic Programming, The MIT Press, Cambridge.

Koontz, W. L. G. and Fukunaga, K. (1972) "Asymptotic analysis of a nonparametric estimate of a multivariate density function", IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol 21: 967-974.

Koza, J. R. (1992) Genetic Programming, The MIT Press, Cambridge.

Lewis, T. G. and Smith M. Z. (1982) Applying Data Structures, Houghton Mifflin Company, Boston.

Linde, Y. and Gray, R. M. (1980) "An algorithm for vector quantizer design", IEEE Transactions on Communications. Vol. COM-28, no. 1: 84-95.

McClelland, J. L. and Rumelhart D. E. (1989) Explorations In Parallel Distributed Processing (A Handbook of Models, Programs, and Exercises), The MIT Press, Cambridge.

Press, W. H. (and others) (1992) Numerical Recipes In C (2nd Edition), Cambridge University Press, Cambridge.

Rumelhart, D. E. (and others) (1994) "The basic ideas in Neural Networks", Communications of the ACM, Vol. 37, no. 3: 87-92.

Schalkoff, R. J. (1992) Pattern Recognition - Statistical, Structural And Neural Approaches, John Wiley & Sons, New York.

Specht D. (1988) "Probabilistic neural networks for classification, mapping, or associative memory", Proceedings of the IEEE International Conference on Neural Networks, Vol 1: 525-532.

Specht D. (1990) "Probabilistic neural networks", Neural Networks, Vol 3: 108-109.

Specht D. (1991) "Generalization accuracy of probabilistic neural networks compared with back-propagation networks", Proceedings of the International Joint Conference on Neural Networks, Vol 1: 887-892.

Tackett, W. A. (1993) "Genetic programming for feature discovery and image discrimination", Proceedings of the Fifth International Conference on Genetic Algorithms: 303 - 309.