A Comparative Study of the Practical Characteristics of Neural Network and Conventional Pattern Classifiers

by

Kenney Ng

Submitted to the Department of Electrical Engineering and Computer Science in Partial Fulfillment of the Requirements for the Degrees of Master of Science and Bachelor of Science at the Massachusetts Institute of Technology May 1990

© Massachusetts Institute of Technology, 1990

Signature of Author __________________________________________

Department of Electrical Engineering and Computer Science
June 1990

Certified by ________________________

Dr. Richard P. Lippmann
Research Staff, MIT Lincoln Laboratory
Thesis Supervisor

Certified by ________________________

Dr. Victor W. Zue
Principal Research Scientist, EECS
Thesis Supervisor

Accepted by ________________________

Arthur C. Smith
Chairman, Department Committee on Graduate Students

ARCHIVES

AUG 10 1990
A Comparative Study of the Practical Characteristics
of Neural Network and
Conventional Pattern Classifiers

by

Kenney Ng

Submitted to the Department of Electrical Engineering and
Computer Science on June 1990 in partial fulfillment of the
requirements for the Degrees of Master and Bachelor of Science

Abstract

Six different pattern classifiers were implemented on a serial computer and com-
pared using artificial and speech recognition tasks. Two neural network classifiers
(radial basis function and high order polynomial GMDH network), and four more
conventional classifiers (Gaussian mixture, linear tree, KD-tree, and condensed K
nearest neighbor) were investigated. These classifiers were chosen to be representative
of different approaches to the task of pattern recognition, and to complement and
extend those that were investigated in a previous study [12]. The goal was to analyze
and compare the performance and behavior of the classifiers on different tasks in
terms of classification error rate, complexity, memory requirements, and training and
classification times.

Radial basis function classifiers generalized well to high dimensional spaces, and
provided low error rates with training times that were much less than those of
back-propagation classifiers. High order polynomial classifiers provided intermediate
error rates, but often required very long training times. In addition, the decision
regions formed did not extrapolate well to data poor regions of the input space.
Gaussian mixture classifiers provided good performance when the numbers and types
of mixtures were selected carefully to model class densities well. Linear tree classifiers
were the most computationally efficient, but performed poorly for high dimensionality
and small training set size. KD-tree classifiers reduced classification time by a factor
of four over conventional K nearest neighbor (KNN) classifiers for low two-input
dimension problems, but provided little or no reduction in classification time for
high 22-input dimension problems. Condensed KNN classifiers reduced memory
requirements over conventional KNN classifiers by a factor of two to fifteen for all
problems, without increasing the error rate significantly.
A new multi-scale radial-basis function classifier was also investigated. It has multiple radial-basis functions centered on each cluster center with widths that vary over a wide range. These classifiers provide error rates that are similar to those of more conventional radial basis function classifiers but eliminate the need for cross-validation testing to select basis function widths.

The results of these two studies demonstrate that classification error rates can be equivalent across different classifiers when they are powerful enough to form minimum error decision regions, when they are properly tuned, and when sufficient training data is available. Practical characteristics such as training time, classification time, and memory requirements, however, differed by orders of magnitude. These results suggest that the selection of a classifier for a particular task should be guided, not just by the error rate, but by practical considerations concerning memory usage, computational resources, and restrictions on training and classification times.

Thesis Supervisor: Dr. Richard P. Lippmann
Title: Research Staff, MIT Lincoln Laboratory

Thesis Supervisor: Dr. Victor W. Zue
Title: Principal Research Scientist, EECS
Acknowledgements

I would like to thank Richard Lippmann for his guidance and support throughout the work. In addition, I would like to thank him for reading and editing the thesis.

I would like to thank Victor Zue for his interest and support.

I would like to thank William Huang and Yuchun Lee for their help.

I would like to thank everyone in the Speech Systems Technology Group at Lincoln Laboratory for their help and support.
Contents

Abstract

1 Introduction
  1.1 Thesis Overview

2 Classification Algorithms
  2.1 Radial Basis Function Classifier
  2.1.1 Network Structure
  2.1.2 Training Algorithm
  2.2 High Order Polynomial Network Classifier
  2.2.1 Network Structure
  2.2.2 Training Algorithm
  2.2.3 Error Criterion
  2.3 Linear Decision Tree Classifier
  2.3.1 Training Algorithm
  2.4 Gaussian Mixture Classifier
  2.4.1 Training Algorithm
  2.5 K-Nearest Neighbor Classifiers
  2.5.1 Condensed K-Nearest Neighbor Classifier
  2.5.2 KD Tree Classifier
  2.6 Chapter Summary

3 Data Sets
  3.1 Bullseye Classification Problem
  3.2 Disjoint Classification Problem
  3.3 F1-F2 Vowel Recognition Problem
  3.4 Digit Recognition Problem

4 Experimental Results
  4.1 Radial Basis Function Classifier
  4.1.1 Optimal Width of Basis Functions
  4.1.2 Optimal Number of Basis Functions
  4.1.3 Determining the Basis Function Centers
  4.1.4 Multi-Scale Basis Functions
  4.2 High Order Polynomial Network Classifier
  4.2.1 Determining Appropriate Polynomial Functions

3
13
15
16
16
17
20
23
23
25
30
33
35
43
45
48
49
52
56
57
57
59
60
61
63
64
64
68
71
74
80
80
4.2.2 PSE Error Function ........................................ 80
4.2.3 Regularity Criterion Error Function .......................... 89
4.2.4 Comparison of the PSE and Regularity Error Functions .......................... 95

4.3 Linear Decision Tree Classifier .................................... 97
4.3.1 Number of Nodes .................................................. 99

4.4 Gaussian Mixture Classifier ......................................... 100
4.4.1 Diagonal and Full Covariance Matrices ......................... 102
4.4.2 Number of Mixture Densities ................................ 106

4.5 Condensed K Nearest Neighbor Classifier .......................... 108
4.5.1 Order of Presentation ........................................ 108
4.5.2 Number of Nearest Neighbors ................................ 114

4.6 KD Tree Classifier .................................................. 116
4.6.1 Bucket Size ...................................................... 116
4.6.2 Number of Nearest Neighbors ................................ 118

4.7 Chapter Summary .................................................. 119

5 Classifier Comparison and Discussion .................................. 121
5.1 Decision Regions and Feature Space Partitioning .................. 124
5.2 Classification Error Rate ........................................... 130
5.2.1 McNemar’s Significance Test .................................. 130
5.2.2 Test Set Error Rate ........................................... 132
5.2.3 Training Set Error Rate ........................................ 138

5.3 Memory Usage ..................................................... 140
5.3.1 Training Memory Usage ........................................ 140
5.3.2 Classification Memory Usage .................................. 142

5.4 Training and Classification Time .................................... 144

5.5 Trade-Offs Between Performance Criteria .......................... 148
5.5.1 Classification Time vs. Classification Memory .................. 148
5.5.2 Training Time vs. Classification Memory ........................ 150

5.6 Chapter Summary .................................................. 152

6 Summary and Future Directions ....................................... 153
## List of Figures

2.1 The Radial Basis Function Classifier. .......................... 17  
2.2 The Radial Basis Function Classifier Training Algorithm. 21  
2.3 The Radial Basis Function Classifier Classification Algorithm. 22  
2.4 The High Order Polynomial (GMDH) Classifier. .......... 24  
2.5 The High Order Polynomial Network Classifier Training Algorithm. 26  
2.6 The High Order Polynomial Network Classifier Classification Algorithm. 27  
2.7 Predicted Square Error (PSE). ............................... 32  
2.8 Linear Decision Tree Classifier. ............................... 34  
2.9 The Linear Decision Tree Classifier Training Algorithm. 36  
2.10 The Linear Decision Tree Classifier Classification Algorithm. 37  
2.11 Decision Regions formed by the Linear Tree Classifier on the 10 Class,  
2-Dimensional Vowel Problem. ................................ 38  
2.12 Two Class Gaussian Mixture Distribution. ................. 45  
2.13 The Gaussian Mixture Classifier Training Algorithm. .... 46  
2.14 The Gaussian Mixture Classifier Classification Algorithm. 47  
2.15 The Condensed K-Nearest Neighbor Classifier Training Algorithm. 50  
2.16 The Condensed K-Nearest Neighbor Classifier Classification Algorithm. 50  
2.17 A Stylized Two-Dimensional Two-Class Distribution Illustrating the  
Data Vectors Stored by the Condensed Nearest Neighbor Classifier. 51  
2.18 The KD Tree (KNN) Classifier Training Algorithm. ........ 52  
2.19 The KD Tree (KNN) Classifier Classification Algorithm. .. 53  
2.20 A Stylized Two-Dimensional Two-Class Distribution Illustrating the  
Data Vectors Searched by the KD Tree KNN Classifier. ........ 55  

3.1 Training Set for the Bullseye Problem. ....................... 58  
3.2 Training Set for the Disjoint Problem. ....................... 59  
3.3 Training Set for the Vowel Problem. ......................... 60  
3.4 Extraction of Cepstral Coefficients for the Digit Problem. ... 61  

4.1 Error Rate of the RBF Classifier on the Bullseye Problem as a Function  
of $H$ and the Number of Basis Function Nodes. ............... 65  
4.2 Error Rate of the RBF Classifier on the Vowel Problem as a Function  
of $H$ and the Number of Basis Function Nodes. ............... 67  
4.3 Error Rate of the RBF Classifier on (A) the Test Set and (B) the  
Training Set of the Digit Problem as a Function of $H$ and the Number  
of Basis Function Nodes. .................................... 69
4.4 Error Rate of the RBF Classifier on the Vowel Problem For Different Methods of Determining Basis Function Centers. .......................... 72
4.5 Error Rate of the RBF Classifier on the Digit Problem For Different Methods of Determining Basis Function Centers. ......................... 73
4.6 Error Rate of the Multiscale and Normal RBF Classifiers on the Disjoint Problem as a Function of the Number of Basis Function Centers. 76
4.7 Error Rate of the Multiscale and Normal RBF Classifiers on the Vowel Problem as a Function of the Number of Basis Function Centers. .... 77
4.8 Error Rate of the Multiscale and Normal RBF Classifiers on (A) the Test Set and (B) the Training Set of the Digit Problem as a Function of the Number of Basis Function Centers. ..................... 79
4.9 Error Rate of the GMDH-PSE Classifier on the Disjoint Problem as a Function of the Complexity of the Classifier. .............................. 81
4.10 Error Rate of the GMDH-PSE Classifier on the Vowel Problem as a Function of the Complexity of the Classifier. ............................. 82
4.11 (A) The Predicted Squared Error and (B) Error Rate of the GMDH Classifier on the Vowel Problem as a Function of the Number of Layers. 85
4.12 Error Rate of the GMDH Classifier on the Disjoint Problem as a Function of the Width of Each Layer. ........................................ 87
4.13 Error Rate of the GMDH Classifier on the Vowel Problem as a Function of the Width of Each Layer. ........................................ 88
4.14 (A) The Regularity Error (rmin) and (B) the Error Rate of the CV GMDH Classifier on the Vowel Problem as a Function of the Number of Layers. ......................................................... 91
4.15 Error Rate of the CV GMDH Classifier on the Vowel Problem as a Function of the Width of Each Layer. ....................................... 94
4.16 Error Rates of GMDH Classifiers Designed Using the CV and the PSE Error Criterion on the Digit Problem as a Function of the Number of Variables at Each Layer. ........................................ 96
4.17 Error Rates of GMDH Classifiers Designed Using the CV and the PSE Error Criterion on the Vowel Problem as a Function of the Number of Layers. ......................................................... 96
4.18 Error Rate of the Linear Tree Classifier on the Disjoint Problem as a Function of the Number of Nodes in the Tree. ............................. 97
4.19 Error Rate of the Linear Tree Classifier on the Vowel Problem as a Function of the Number of Nodes in the Tree. ............................. 98
4.20 Stopping Point Predicted by the 1 SE Rule for the Vowel Problem. .... 100
4.21 Error Rate of the Gaussian Mixture Classifier for the Bullseye Problem Using (A) Diagonal Covariance Estimates and (B) Full Covariance Estimates. ......................................................... 103
<table>
<thead>
<tr>
<th>Figure Number</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.22</td>
<td>Error Rate of the Gaussian Mixture Classifier for the Digit Problem</td>
<td>107</td>
</tr>
<tr>
<td></td>
<td>Using (A) Diagonal Covariance Estimates and (B) Full Covariance Estimates.</td>
<td></td>
</tr>
<tr>
<td>4.23</td>
<td>Error Rate of the Condensed Nearest Neighbor Classifier on the Vowel Problem</td>
<td>109</td>
</tr>
<tr>
<td></td>
<td>as a Function of the Number of Nearest Neighbors for 10 Randomly Initialized Runs.</td>
<td></td>
</tr>
<tr>
<td>4.24</td>
<td>Set of Vectors Selected by the Condensed Nearest Neighbor Classifier</td>
<td>110</td>
</tr>
<tr>
<td></td>
<td>for the Bullseye Problem.</td>
<td></td>
</tr>
<tr>
<td>4.25</td>
<td>Error Rate of the Condensed Nearest Neighbor Classifier on (A) the Disjoint Problem and (B) the Vowel Problem as a Function of the Number of Nearest Neighbors for Three Different Training Methods.</td>
<td>113</td>
</tr>
<tr>
<td>4.26</td>
<td>Error Rate of the Condensed Nearest Neighbor Classifier on the Vowel Problem as a Function of ( K' ).</td>
<td>115</td>
</tr>
<tr>
<td>4.27</td>
<td>(A) Number of Exemplars Examined and (B) Time Required by the KD Tree Classifier for the Disjoint Problem as a Function of the Bucket Size.</td>
<td>117</td>
</tr>
<tr>
<td>4.28</td>
<td>Error Rate of the KD Tree Classifier on (A) the Vowel Problem and (B) the Disjoint Problem as a Function of the Number of Nearest Neighbors.</td>
<td>119</td>
</tr>
<tr>
<td>5.1</td>
<td>Decision Regions Formed by the RBF Classifier for the Vowel Problem.</td>
<td>125</td>
</tr>
<tr>
<td>5.2</td>
<td>Decision Regions Formed by the Gaussian Mixture Classifier for the Vowel Problem.</td>
<td>126</td>
</tr>
<tr>
<td>5.3</td>
<td>Decision Regions Formed by the Linear Tree Classifier for the Vowel Problem.</td>
<td>127</td>
</tr>
<tr>
<td>5.4</td>
<td>Decision Regions Formed by the GMDH Classifier for the Vowel Problem.</td>
<td>128</td>
</tr>
<tr>
<td>5.5</td>
<td>Decision Regions Formed by the KD Tree Classifier for the Vowel Problem.</td>
<td>129</td>
</tr>
<tr>
<td>5.6</td>
<td>Test Set Error Rates of the Different Classifiers on the Four Problems.</td>
<td>133</td>
</tr>
<tr>
<td>5.7</td>
<td>Training Set Error Rates of the Different Classifiers on the Four Problems.</td>
<td>139</td>
</tr>
<tr>
<td>5.8</td>
<td>Training Memory Usage for the Different Classifiers on the Four Problems.</td>
<td>140</td>
</tr>
<tr>
<td>5.9</td>
<td>Classification Memory Usage for the Different Classifiers on the Four Problems.</td>
<td>143</td>
</tr>
<tr>
<td>5.10</td>
<td>Training Times in CPU Seconds for the Different Classifiers.</td>
<td>145</td>
</tr>
<tr>
<td>5.11</td>
<td>Classification Times in CPU Milliseconds for the Different Classifiers.</td>
<td>147</td>
</tr>
<tr>
<td>5.12</td>
<td>Classification Time versus Classification Memory Usage for the Vowel Problem.</td>
<td>149</td>
</tr>
<tr>
<td>5.13</td>
<td>Training Time versus Classification Memory Usage for the Vowel Problem.</td>
<td>151</td>
</tr>
</tbody>
</table>
List of Tables

4.1 Number of Basis Functions and Values of $H$ Chosen for the R3F Classifiers for the Four Problems. ........................................ 71
4.2 Estimates of $\sigma^2$ Determined Using a KNN Classification on the Training Set and the Resulting Number of Coefficients in the Final GMDH Output Polynomials. ........................................ 84
4.3 Determining The Proper Complexity for the CV GMDH Classifier. .. 92
4.4 Error Rates for the CV GMDH and “All Data” GMDH Classifiers. ... 93
4.5 Number of Training Vectors Stored by the Condensed $K$ Nearest Neighbor Classifier for Different Methods of Training. .......... 111

5.1 Summary of Parameter Values and Sizes for Selected Classifiers. ... 122
5.2 Performance Summary of the Selected Classifiers. ................... 123
5.3 Statistical Significance ($P$) of Test Set Error Rate Differences for the Bullseye Problem (Asterisks Indicate Differences that are Statistically Significant at the 0.01 Level). ................................. 134
5.4 Statistical Significance ($P$) of Test Set Error Rate Differences for the Disjoint Problem. .............................................. 135
5.5 Statistical Significance ($P$) of Test Set Error Rate Differences for the Vowel Problem (Asterisks Indicate Differences that are Statistically Significant at the 0.01 Level). ................................. 135
5.6 Statistical Significance ($P$) of Test Set Error Rate Differences for the Digit Problem (Asterisks Indicate Differences that are Statistically Significant at the 0.01 Level). ................................. 136
Chapter 1

Introduction

The goal of pattern recognition or pattern classification is to classify observed input patterns/vectors into one of several different classes. The definition of a class is variable and is usually defined with respect to a particular application. For example, in speech recognition, pattern classification may involve classifying temporally varying spectral patterns representing spoken words. In this case, each class could represent a word to be recognized. However, to identify individual speech sounds, each class could represent an individual phoneme. To classify speech sounds in terms of distinctive features, the classes would be the set of distinctive features [7] including HIGH, BACK, ROUND, etc.

In general, a pattern recognition system has two stages, a pre-processing stage, and a classification stage. The pre-processing stage reduces the complexity of the input space by mapping raw inputs into input patterns or feature vectors in a feature space. This mapping is pre-designed with a specific application in mind and usually corresponds to extracting “important” and “useful” features from more complex inputs. For example, a typical preprocessor for speech extracts cepstral coefficients, which are derived from a spectral analysis of short segments of the input speech waveform. These coefficients provide good classification and are used widely in the speech recognition community [19]. A wide range of other types of “feature vectors,” can be constructed depending on the particular task. Some other possible speech pre-processing methods include extracting formant frequencies, and using physiologically-based methods that extract phase or synchrony information [17].

After raw inputs are transformed into input patterns or feature vectors, a classifier maps these feature vectors into the set of different classes. This mapping or classification results in the partitioning and labeling of regions in the feature space. Each vector in the feature space is assigned to be a member of a certain class. Unlike
the pre-processor, the classifier is data driven, or adaptive. It can learn the mapping function from feature space to class space given labeled training examples.

Recently, there has been a resurgence of interest in adaptive pattern classifiers. A good review is provided in [14]. Although classifiers have been analyzed in the past, very few studies have actually compared the performance of many different classifiers in practical terms. Most of the studies involve only one or two classifiers, and a set of data for training and testing. These studies concentrate heavily on the accuracy of classification and not enough on other issues such as ease of implementation, complexity of the classification algorithm, memory and time requirements, and relative performance of other classifiers. The reason for this over-emphasis probably lies in a common belief that one classifier will be able to classify better than others regardless of implementation details. A major hypothesis of this work is that given enough resources for implementation and sufficient training data, most modern classifiers can perform equally well.

Research in pattern classification should emphasize the analysis of how classifiers perform under implementation constraints. Because the performance of a classifier depends on the characteristics of the problem it is applied to, it is important to understand first what factors are important and second, how these factors affect the efficiency and performance of a classifier. In addition, the importance of studying a number of different classifiers jointly should be stressed in order to help develop guidelines for the selection and use of classifiers.

The focus of this work is to study the behavior and compare the performance and practical characteristics of several different classifiers on artificial and speech recognition tasks. There are two motivations for this study. One is the desire to gain insight into the behavior of different classifiers on speech problems, and the second is to understand performance issues to aid in developing guidelines for selecting and using classifiers given a particular task or set of constraints. Some of the performance issues that were investigated include:

- Classification Error Rate
- Model Complexity
- Memory Requirements
- Training and Classification Times
The classification algorithms included in this study were:

- Radial Basis Function (RBF) Classifiers
- Multi-Scale Radial Basis Function Classifiers
- High Order Polynomial Network (GMDH) Classifiers
- Linear Decision Tree Classifiers
- Gaussian Mixture Classifiers
- KD Tree (KNN) Classifiers
- Condensed K Nearest Neighbor Classifiers

1.1 Thesis Overview

The remainder of this thesis is divided into two parts. The first part is made up of Chapters 2 and 3. Chapter 2 describes the six different classifiers and includes detailed algorithmic descriptions of each. Chapter 3 describes the four different data sets used to evaluate the classifiers. The second part includes Chapters 4 and 5. Chapter 4 discusses the results of the tuning experiments and Chapter 5 compares the performance and practical characteristics of the classifiers on the four problems. A brief summary and some remarks about directions for future research are provided in Chapter 6.
Chapter 2

Classification Algorithms

The radial basis function (RBF) and high order polynomial (GMDH) neural network classifiers, and the Gaussian mixture, linear tree, KD-tree, and condensed nearest neighbor conventional classifiers are described in this chapter. These particular classifiers were chosen to be representative of different approaches to the task of pattern recognition, and to complement those that were studied in [12]. A brief discussion and a detailed step by step algorithmic description is provided for each of the six different classifiers.

2.1 Radial Basis Function Classifier

The method of radial basis functions can be viewed in two different ways. One view is to interpret the RBFs as kernel functions that expand input vectors into a high dimensional space, trying to take advantage of the mathematical fact that a classification problem cast into a high dimensional space is more likely to be linearly separable than one in a low dimensional space [22]. In an RBF classifier, the input vectors are "projected" or expanded onto a high dimensional space defined by the set of basis functions, \( \{ \Phi_i \} \). A linear combination of these basis functions, \( \sum_i w_i \Phi_i \), for some set of weights \( \{ w_i \} \), defines a hyperplane in the high dimensional space that divides the space linearly into two regions. This decision boundary serves to distinguish one class from all others.

Another view is to interpret a RBF classifier as a function-mapping interpolation method that tries to construct hypersurfaces, one for each different class, by taking a linear combination of the basis functions. These hypersurfaces can be viewed as discriminant functions, where the the surface has a "high" value for the class it represents and a "low" value for all the other classes. An unknown input vector is
2.1. RADIAL BASIS FUNCTION CLASSIFIER

![Diagram of Radial Basis Function Classifier]

Figure 2.1: The Radial Basis Function Classifier.

classified as belonging to the class associated with the hypersurface with the \textit{largest output} at that point. In this case the basis functions do not serve as a \textit{basis} for a high dimensional space, but as \textit{components} in a "finite expansion" of the desired hypersurface, where the component coefficients, ie. the weights, have to be trained.

2.1.1 Network Structure

A RBF classifier has an architecture very similar to that of the traditional 3-layer back-propagation multi-layer perceptron network [13, 24]. Figure 2.1 shows a RBF classifier with five input nodes, four hidden basis function nodes, and three output nodes. Connections between the input and middle layers have \textit{unit} weights and, as a result, do not have to be trained. Nodes in the middle layer, called basis function (BF) nodes, have a Gaussian pulse nonlinearity specified by a particular mean vector, $\mu_i$, and variance vector, $\sigma^2_i$, where $i = 1, \ldots, F$ and $F$ is the number of basis function nodes. Note that $\sigma^2_i$ represents the \textit{diagonal} entries of the \textit{covariance} matrix of Gaussian pulse $i$. Given a $D$-dimensional input vector, $X$, each basis function node $i$ outputs a \textit{scalar} value, $y_i$, reflecting the activation of the basis function caused by
that input:

$$y_i = \Phi_i(||X - \mu||) = \exp \left[ - \sum_{k=1}^{D} \frac{(x_k - \mu_{ik})^2}{2h\sigma^2_{ik}} \right],$$

(2.1)

where $H$ is a proportionality constant for the variance, $x_k$ is the $k^{th}$ component of the input vector $X = [x_1, x_2, \ldots, x_D]$, and $\mu_{ik}$ and $\sigma^2_{ik}$ are the $k^{th}$ components of the mean and variance vectors, respectively, of basis function node $i$. Inputs that are "close," in a Euclidean sense, to the center of the Gaussian basis function will result in higher activations while those that are far away will result in lower activations.

Since each output node of the RBF network forms a linear combination of the basis function node activations, the network connecting the middle and output layers is a linear network:

$$z_j = \sum_{i} w_{ij}y_i + w_{0j}$$

(2.2)

where $z_j$ is the output of the $j^{th}$ output node, $y_i$ is the activation of the $i^{th}$ basis function node, $w_{ij}$ is the weight connecting the $i^{th}$ basis function node to the $j^{th}$ output node, and $w_{0j}$ is the bias or threshold of the $j^{th}$ output node. This bias comes from weights associated with a basis function node (in this case basis function node $i = 0$) that has a constant unit output regardless of the input. An unknown vector, $X$, is classified as belonging to the class associated with the output node, $j$, with the largest output, $z_j$.

The weights, $w_{ij}$, in the linear network do not have to be solved using iterative minimization methods like gradient descent. They can be determined quickly and exactly using a matrix pseudo-inverse approach if the desired outputs are 1 and 0, a squared error cost function is used, and the basis function parameters (i.e., $\mu_i$ and $\sigma^2_i$) are fixed [22, 23]. To illustrate this approach, let the actual RBF classifier output (2.2) for a given input vector $X(p)$, with class label $C(p)$, at output node $j$ be $z_j(p)$, and the desired output be $d_j(p)$, where

$$d_j(p) = \begin{cases} 
1 & \text{if } C(p) = j, \\ 
0 & \text{otherwise}
\end{cases}, \quad j = 1, \ldots, M,$$

(2.3)

$M$ is the number of classes, and $p$ is an index to the input vectors. Let the optimal
weights be defined as those for which the squared error measure

\[ E = \frac{1}{2} \sum_{p=1}^{N} \sum_{j=1}^{M} [d_j(p) - z_j(p)]^2, \]  

(2.4)

or

\[ E = \frac{1}{2} \sum_{p=1}^{N} \sum_{j=1}^{M} \left[ d_j(p) - \sum_{i=0}^{F} w_{ij} y_i(p) \right]^2 \]  

(2.5)

is the smallest. Here, \( N \) is the total number of patterns in the training set.

The minimum error occurs at the point where the derivative of (2.4) or (2.5)

\[ \frac{dE}{dw_{ij}} = \sum_{p=1}^{N} \sum_{j=1}^{M} -y_i(p) \left[ d_j(p) - \sum_{i=1}^{F} w_{ij} y_i(p) \right] \]  

(2.6)

\[ = \sum_{l=1}^{F} w_{lj} \left[ \sum_{p=1}^{N} y_i(p)y_i(p) \right] - \sum_{p=1}^{N} y_i(p)d_j(p) \]  

(2.7)

goes to zero.

Let \( R \) be the \( F \times F \) correlation matrix of the basis function outputs:

\[ R_{il} = \sum_{p=1}^{N} y_i(p)y_i(p) \]  

(2.8)

and \( B \) be the \( F \times M \) output matrix:

\[ B_{lj} = \sum_{p=1}^{N} y_i(p)d_j(p). \]  

(2.9)

Substituting (2.8) and (2.9) into (2.7), yields

\[ \frac{dE}{dw_{ij}} = \sum_{l=1}^{F} w_{lj} [R_{il}] - B_{lj} \]  

(2.10)

and solving this for \( w^* \), the weight matrix that minimizes \( E \), gives:

\[ w^*_{ij} = \frac{F}{l=1} (R^{-1})_{il} B_{lj}, \]  

(2.11)

where \( R^{-1} \) is the matrix inverse of \( R \). Thus, the problem can be solved exactly by inverting a square \( F \times F \) correlation matrix. A variety of methods can be used to invert \( R \). In this work, a singular value decomposition (SVD) was used to perform the matrix inversion. This technique allows checking and handling of singular matrices, and is, in general, more robust than standard Lower-Upper triangular matrix factorization or LU decomposition [21].
2.1.2 Training Algorithm

A detailed algorithmic description of the radial basis function classifier used in this study is shown in Figures 2.2 and 2.3. Initially, the size of the RBF network is determined by selecting $F$, the number of basis functions. The appropriate value of $F$ is problem specific and usually depends on the dimensionality of the problem and the complexity of the decision regions that have to be formed. In general, $F$ can be determined empirically by trying a variety of different $F$’s, or can be set to some constant number, usually larger than the input dimension of the problem. Experiments that examine the effect of $F$ on the error rate for different problems are described in section 4.1.

After the number of basis functions are set, the mean, $\mu_i$, and variance, $\sigma_i^2$, vectors of the basis functions can be determined. A variety of methods can be used to determine these vectors. They can be trained along with the output weights using a back-propagation gradient descent technique, but this usually requires a long training time and may lead to sub-optimal local minima [16]. Alternatively, the means and variances can be determined before training the output weights. Training of the network then would involve only determining the weights, which, as discussed above, can be done with a single matrix inversion.

If the RBF classifier is viewed as a function mapping interpolation, it is clear that the basis function centers and variances should be set so as to “cover” the space of interest. Different techniques have been suggested to select centers and variances that cover this space. One approach is to use a “grid” of equally spaced basis functions that sample the input space [6]. Another approach is to use a clustering algorithm, like k-means or the Estimate-Maximize (E-M) algorithm, to determine the set of basis function centers [8, 16]. Yet another approach is to choose random vectors from the training set as basis function centers, making sure that each class is represented [22]. In section 4.1, experiments comparing different RBF classifiers initialized using k-means determined basis function centers, E-M determined basis function centers, and randomly selected BF centers are described.

Once the basis function centers or means are determined, the variances or widths, $\sigma_i^2$, of the basis functions can be set. They can be fixed to some global value or
2.1. RADIAL BASIS FUNCTION CLASSIFIER

1. Initialization

(a) Fix the network structure by selecting $F$, the number of basis functions, where each basis function, $i$, has the output

$$y_i = \Phi(||X - \mu_i||) = \exp \left[ -\sum_k \frac{(x_k - \mu_{ik})^2}{2H \sigma_{ik}^2} \right],$$

where $k$ is the component index.

(b) Determine the basis function means, $\mu_i$, where $i = 1, \ldots, F$, using k-means clustering, E-M estimation, or by picking random training samples.

(c) Determine the basis function variances, $\sigma_i^2$, where $i = 1, \ldots, F$, by setting $\sigma_i^2$ equal to the average Euclidean distance from $\mu_i$ to its five nearest neighbors.

(d) Determine $H$, a global proportionality factor for the basis function variances by empirical search.

2. Training Presentation

(a) Input training patterns, $X(p)$, and their class labels, $C(p)$, to the classifier, where the pattern index is $p = 1, \ldots, N$.

(b) Compute the output of the basis function nodes, $y_i(p)$, where $i = 1, \ldots, F$, resulting from pattern $X(p)$.

(c) Compute the $F \times F$ correlation matrix, $R$, of the basis function outputs:

$$R_{ii} = \sum_p y_i(p)y_i(p).$$

(d) Compute the $F \times M$ output matrix, $B$, where $d_j$ is the desired output and $M$ is the number of output classes:

$$B_{lj} = \sum_p y_l(p)d_j(p), \text{ where } d_j(p) = \begin{cases} 1 & \text{if } C(p) = j \\ 0 & \text{otherwise} \end{cases}, \text{ and } j = 1, \ldots, M.$$

3. Determine Weights

(a) Invert the $F \times F$ correlation matrix $R$, to get $R^{-1}$.

(b) Solve for the weights in the network using the following equation:

$$w_{lj} = \sum_i (R^{-1})_{li}B_{lj}.$$

Figure 2.2: The Radial Basis Function Classifier Training Algorithm.
1. Present Input Pattern $X_{test}$ to the Classifier.

2. Classify $X_{test}$
   
   (a) Compute the basis function outputs, $y_i = \Phi(||X_{test} - \mu_i||)$, for all $F$ basis functions.
   
   (b) Compute output node activations: $z_j = \sum_i w_{ij} y_i + w_{0j}$
   
   (c) Select the output, $z_j$, with the largest value and classify $X_{test}$ as class $j$.

Figure 2.3: The Radial Basis Function Classifier Classification Algorithm.

can be set to reflect the density of data vectors in the vicinity of the basis function center [16]. By setting the variance of the basis function to the average distance from the basis function center to its nearest $K$ neighbors, or to the variance of the k-means cluster, the basis functions will be narrower in regions of dense population, and wider in regions of sparse population. These locally tuned basis functions may be desirable since finer grain decision boundaries, which can be formed with narrower basis functions, may be required in densely populated regions. Alternatively, the E-M algorithm can be used to determine both the means and the variances simultaneously.

In this study, each basis function variance, $\sigma_i^2$, is set equal to the average Euclidean distance from the basis function center to its five nearest neighbors. This number of nearest neighbors was determined empirically. In addition, a global proportionality factor, $H$, for the basis function variances is included to allow rescaling of the basis function widths. The proper value of $H$ is determined by searching the space of $H$ for values that result in good performance. Experiments used to determine the proper value of $H$ are described in section 4.1.

After the basis function parameters are set, the next step is to train the output weights. To do this, individual training patterns, $X(p)$, and their class labels, $C(p)$, are presented to the classifier and the resulting basis function node outputs, $y_i(p)$ defined in equation 2.1, are computed. These outputs, along with the desired outputs, $d_j(p)$ (equation 2.3), are then used to determine the $F \times F$ correlation matrix $R$.
2.2. HIGH ORDER POLYNOMIAL NETWORK CLASSIFIER

(equation 2.8) and the \( F \times M \) output matrix \( \mathbf{B} \) (equation 2.9). Note that each training pattern produces one \( \mathbf{R} \) and one \( \mathbf{B} \) matrix. The final \( \mathbf{R} \) and \( \mathbf{B} \) matrices are the result of the sum of \( N \) individual \( \mathbf{R} \) and \( \mathbf{B} \) matrices, where \( N \) is the total number of training patterns. Once all \( N \) training patterns have been presented to the classifier, the output weights, \( w_{ij} \) can be determined. The final correlation matrix \( \mathbf{R} \) is inverted and equation 2.11 is used to determine each \( w_{ij} \).

Classification is performed by presenting an unknown input vector, \( \mathbf{X}_{\text{test}} \), to the trained classifier, and computing the resulting basis function node outputs, \( y_i \) (equation 2.1). These values are then used, along with the weights, \( w_{ij} \) to compute the output values, \( z_j \) defined in equation 2.2. \( \mathbf{X}_{\text{test}} \) is then classified as belonging to the class associated with the output node, \( j \), with the largest \( z_j \) output.

2.2 High Order Polynomial Network Classifier

The high order polynomial network is a self-organizing algorithm, that tries to construct a set of high-order polynomial discriminant functions in the feature space that can be used to discriminate between the different classes [1, 2, 3]. This algorithm is also known as the Group Method of Data Handling (GMDH). It grew out of cybernetics research and was introduced in the mid 1960's by the Soviet scientist A. G. Ivakhnenko [9].

The idea behind the high order polynomial network is to adaptively construct a polynomial for each of the different classes, and then to classify an unknown input vector as belonging to the class associated with the polynomial with the largest output caused by that vector.

2.2.1 Network Structure

The GMDH algorithm forms a complex, multi-layer, high-order polynomial network out of simpler function nodes that form first, second, and third order polynomials. These polynomials are derived from regression equations of inputs taken one or two at a time from the previous layer of the network. Figure 2.4 shows a partial
GMDH network structure with three network layers. The first layer is made up of the components of the input vectors. This means that if the input vectors are $D$-dimensional, then the first layer will have $D$ variables. The second layer is made up of polynomials, $P_i(\cdot)$, that take as input all possible combinations of the variables in the first layer taken one or two at a time. With $D$ variables in the first layer, there will be $Q = \frac{D(D-1)}{2} + D$ newly constructed polynomials in the second layer. The first term, $\frac{D(D-1)}{2}$, comes from taking all possible combinations of $D$ variables two at a time and the second term, $D$, comes from taking all possible combinations of $D$ variables one at a time. The third layer is also made up of polynomials of the same form as those in the second layer, but now they take as input all $\frac{Q(Q-1)}{2} + Q$ possible combinations of the variables in the second layer taken one or two at a time. Each of these polynomials corresponds to a “node” in the network, and the width of each layer is defined as the number of nodes in the layer. In addition, the original input variables can be fed forward into the third layer, in which case the total number of nodes in the third layer would then be $\frac{Q(Q-1)}{2} + Q + D$.

The types of polynomials that can be used to form the nodes in the GMDH network can vary widely and usually depend on the particular problem. Traditionally, simple first, second, and third order polynomials of one, two, or three input variables are
used. In this study, a variety of different high order polynomials were explored:

\[ P(v_1) = w_0 + w_1v_1 + w_2v_1^2 + w_3v_1^3 \]  
(2.12)

\[ P(v_1, v_2) = w_0 + w_1v_1 + w_2v_2 + w_3v_1v_2 + w_4v_1^2 + w_5v_2^2 + w_6v_1^3 + w_7v_2^3 \]  
(2.13)

\[ P(v_1, v_2, v_3) = w_0 + w_1v_1 + w_2v_2 + w_3v_3 + w_4v_1v_2 + w_5v_3 + w_6v_2v_3 + w_7v_1^2 + w_8v_2^2 + w_9v_3^2 + w_{10}v_1v_2v_3 + w_{11}v_1^3 + w_{12}v_2^3 + w_{13}v_3^3. \]  
(2.14)

Polynomial 2.12 takes one input, \( v_1 \), while 2.13 takes two inputs, \( v_1 \) and \( v_2 \). Compared with 2.12 and 2.13, polynomial 2.14, which takes three input variables, was found to be used very seldomly or not even at all in the final network solutions. It did not improve performance but did increase training time dramatically. The reason for the large increase in training time is because with a three-input polynomial, instead of considering just \( \frac{Q(Q-1)}{2} + Q \) new polynomials at each layer, \( \frac{Q(Q-1)(Q-2)}{6} + \frac{Q(Q-1)}{2} + Q \) new polynomials have to be considered. For large \( Q \), this difference becomes almost a \( Q \)-fold increase in the number of polynomials.

In addition to these “pure” polynomials, other functions, including a cube root and exponential were explored.

\[ P(v_1) = \sqrt[3]{w_0 + w_1v_1} \]  
(2.15)

\[ P(v_1) = \exp(w_0 + w_1v_1) \]  
(2.16)

These functions, like polynomial 2.14, were not used very often in final network solutions and did not improve the performance. As a result, only polynomials 2.12 and 2.13 were used in the GMDH networks.

### 2.2.2 Training Algorithm

A detailed algorithmic description of the GMDH classifier used in this study is shown in Figures 2.5 and 2.6. Since the GMDH classifier requires that one polynomial network be constructed for each of the \( M \) classes in a problem, the first step is to select the first class to be modeled. Once the class is selected, let the input vectors
1. Initialization

(a) Select one of the $M$ classes in the problem to be modeled.
(b) Input training patterns, $X(p)$, and their class labels $C(p)$, where $p = 1, \ldots, N$.
(c) Determine the vector of desired outputs or dependent observations, $D$:
   \[
   d(p) = \begin{cases} 
   1 & \text{if } C(p) \text{ is the class being modeled} \\
   0 & \text{otherwise}
   \end{cases}
   \]
(d) Let the input patterns $X(p)$, $p = 1, \ldots, N$, become the first set of independent observations, $O_1$, $l = 1$.
(e) Let the components of $X(p)$ be set of the independent variables, $V_l$, $l = 1$, of the first layer.

2. Adaptively Build Network Layers

(a) Take all possible combinations of the current independent variables $V_l$ 1 or 2 at a time and construct all regression polynomials using the following polynomial functions, where $w_i$ are unknown coefficients and $v_k$ are components in $V_l$:
   \[
   P(v_1) = w_0 + w_1v_1 + w_2v_1^2 + w_3v_1^3
   \]
   \[
   P(v_1, v_2) = w_0 + w_1v_1 + w_2v_2 + w_3v_1v_2 + w_4v_1^2 + w_5v_2^2 + w_6v_1^3 + w_7v_2^3.
   \]
(b) Solve for $w_i$ with pseudo-inverse, using observations $O_l$ and desired outputs $D$.
(c) For each of the newly constructed $Q$ regression polynomials, evaluate the polynomial $P(\cdot)$ for all $N$ observations in $O_l$ to obtain the new set $O_{l+1}$:
   \[
   o_{l+1}(p) = P(o_l(p)), \text{ for } p = 1, \ldots, N.
   \]
(d) Compute an error function for each of the $Q$ candidate polynomials.
   \[
   \mathcal{E} = \frac{1}{N} \sum_{p=1}^{N} [d(p) - o_{l+1}(p)]^2 + \frac{2}{N} \sigma_p^2 W,
   \]
   where $\sigma_p^2$ is an estimate of the error variance and $W$ is the number of coefficients in the polynomial.
(e) Rank the $Q$ polynomials according to $\mathcal{E}$ and select the $Q_o$ with the lowest error to become $V_{l+1}$, the inputs to the next layer.
(f) Set $\mathcal{E}_{l+1}^* \equiv$ the lowest error of the current layer.
(g) If $\mathcal{E}_{l+1}^* > \mathcal{E}_l^*$, then choose the polynomial from the previous layer with error $\mathcal{E}_l^*$ as the output of the network, $P_j(\cdot)$, and stop.
   Otherwise append the original inputs, $V_l$, to the $Q_o$ polynomials as independent variables and goto 2a.

3. Repeat Steps 1 and 2 for All $M$ Classes, Building One Polynomial Network, $P_j(\cdot)$, for Each Class.

Figure 2.5: The High Order Polynomial Network Classifier Training Algorithm.
2.2. **HIGH ORDER POLYNOMIAL NETWORK CLASSIFIER**

1. Present Input Pattern $X_{test}$ to the Classifier.

2. Classify $X_{test}$
   
   (a) Evaluate the output of each polynomial: $P_j(X_{test})$.
   
   (b) Pick the polynomial, $P_j^*(X_{test})$, with the largest output and classify $X_{test}$ as class $j^*$.

---

Figure 2.6: The High Order Polynomial Network Classifier Classification Algorithm.

of the training set be denoted $X(p)$ and their class labels $C(p)$ for $p = 1, \ldots, N$, where $N$ is the total number of training vectors. Let the $N$ training vectors, $X(p)$, be the first set of independent observations, $O_l$, where $l = 1$ implying that these observations are associated with layer 1. Define a vector of $N$ desired outputs or dependent observations, $D$, where the components

$$
    d(p) = \begin{cases} 
    1 & \text{if } C(p) \text{ is the class being modeled} \\
    0 & \text{otherwise.}
    \end{cases}
$$

(2.17)

The first step in the algorithm is to construct the first layer of the network which, as mentioned earlier, is made up of the $D$ components of the original input vectors, $X(p)$. Let this set of variables in the first layer of the network be denoted by $V_l$, where $l = 1$. The second layer of the network is then constructed by forming polynomials of the form 2.12 and 2.13 using all possible combinations of the current independent variables $V_1$ taken one or two at a time. As before, with $D$ components in the first layer, there will be $Q = \frac{D(D-1)}{2} + D$ polynomials in the second layer.

For each newly formed regression polynomials, $P(\cdot)$, in the second layer, the $K$ unknown coefficients, $w_i, i = 1, \ldots, K$ are solved using the current set of observations $O_1$, and the vector of desired outputs, $D$, by a pseudo-inverse technique very similar to the method used in the radial basis function case. Determining the $w_i$'s essentially involves solving an over-determined system with more equations than unknowns:

$$
    Aw = D.
$$

(2.18)
The $K$-dimensional vector $\mathbf{w}$ is a vector of the unknown coefficients, $w_i$. The $N$-dimensional vector $\mathbf{D}$ contains zeros and ones corresponding to the desired output $d(p)$. The entries in row $p$ of the $N \times K$ matrix, $\mathbf{A}$, are the “combinations” of the input variables, as defined in polynomial $P(\cdot)$, derived from input vector $\mathbf{X}(p)$. For example, the over-determined system for polynomial 2.12 would look like:

$$
\begin{bmatrix}
1 & v_1 & v_1^2 & v_1^3 \\
1 & v_2 & v_2^2 & v_2^3 \\
\vdots \\
1 & v_i & v_i^2 & v_i^3 \\
\vdots \\
1 & v_N & v_N^2 & v_N^3
\end{bmatrix}
\begin{bmatrix}
w_0 \\
w_1 \\
\vdots \\
w_2 \\
\vdots \\
w_3
\end{bmatrix}
= 
\begin{bmatrix}
d(0) \\
d(1) \\
\vdots \\
d(i) \\
\vdots \\
d(N)
\end{bmatrix}.
$$

Thus, the polynomial coefficients can be solved by performing a pseudo-inverse:

$$
\mathbf{w} = \left(\mathbf{A}^T \mathbf{A}\right)^{-1} \mathbf{A}^T \mathbf{D}.
$$

The pseudo-inverse used here is the same as the one used in the RBF classifier. Specifically, a singular value decomposition (SVD) is performed on the matrix $\mathbf{A}^T \mathbf{A}$ and the resulting factored matrices are then used to compute the matrix inverse.

After determining the coefficients of the newly constructed regression polynomials, the next step is to evaluate each polynomial at all the training data vectors to obtain a set of new observations. For example, suppose $v_k(p)$ for $p = 1, \ldots, N$ and $k = 1, \ldots, D$, are the observations in $O_1$ and the first regression polynomial function of the current (second) layer is

$$
P_1(v_1, v_2) = w_0 + w_1 v_1 + w_2 v_2 + w_3 v_1 v_2 + w_4 v_1^2 + w_5 v_2^2 + w_6 v_1^3 + w_7 v_2^3
$$

then the new observations, $z_1(p)$, which will become inputs to the next (third) layer,
2.2. **HIGH ORDER POLYNOMIAL NETWORK CLASSIFIER**

evaluated at all \( N \) training patterns would be:

\[
\begin{align*}
z_1(1) &= P_1(v_1(1), v_2(1)) \\
z_1(2) &= P_1(v_1(2), v_2(2)) \\
& \vdots \\
z_1(N) &= P_1(v_1(N), v_2(N)).
\end{align*}
\]

If the second regression polynomial function is

\[
P_2(v_1, v_3) = w_0 + w_1v_1 + w_2v_3 + w_3v_1v_3 + w_4v_1^2 + w_5v_3^2 + w_6v_1^3 + w_7v_3^3
\]

(2.22)

then the new observations, \( z_2(p) \) for \( p = 1, \ldots, N \), would be:

\[
\begin{align*}
z_2(1) &= P_2(v_1(1), v_3(1)) \\
z_2(2) &= P_2(v_1(2), v_3(2)) \\
& \vdots \\
z_2(N) &= P_2(v_1(N), v_3(N)).
\end{align*}
\]

The remaining new observations, \( z_k(p) \) for \( k = 1, \ldots, Q \), and \( p = 1, \ldots, N \) resulting from all \( Q \) regression polynomials would be computed in a similar manner. These new or "transformed" observations will become the training set for the next layer and are labeled \( O_2 \). These new observations are designed to have better "predictability" powers than \( O_1 \) [9].

In general, the number of regression polynomials grows *exponentially* with the number of layers. In order to keep the computation tractable, the maximum number of variables in each layer or the *width* of the network must be constrained. To do this, an *error function* is evaluated for each of the newly formed regression polynomials. The polynomials are then ranked according to this error measure, and the "best" candidate polynomials (ie. those with the lowest error) are selected to become \( V_2 \), the set of inputs to the next layer. In addition to the selected polynomial variables, the *original* input variables, \( V_1 \), are fed forward at each layer to also become inputs
to the next layer. The availability of the original input variables at every layer has been shown to improve the performance of the classifier [2].

The network continues to grow in this layer by layer fashion until the the lowest error measure of the new layer becomes higher than the lowest error measure of the current layer. At this point, where the network has "converged," the single polynomial at the current layer with the lowest error is selected to be the final output polynomial of the network for the class. In Figure 2.4, the final network solution or output polynomial is the shaded node in the third layer.

Because only one polynomial at the top layer is used, and there is no way in advance of knowing what that polynomial will be, the algorithm must store all the polynomials in the lower layers that combine to form polynomials in the higher layers until the network stops growing. Only then can it back trace down the network marking the specific polynomials that contribute to the final output of the network. In Figure 2.4, the shaded nodes in layers one and two represent the contributing polynomials traced from the final output polynomial. In general, the number of polynomials actually used in the final output is very small compared to the number of polynomials that have to be evaluated and stored. For this reason, a large amount of memory is required during training especially for large input dimensions and for very complex polynomials with many layers.

As mentioned above, one polynomial network is constructed for each of the M classes in the problem. For a multi-class problem, this means that many different polynomials will have to be constructed, possibly requiring a long and computationally intensive training process. After the M output polynomials are constructed, an unknown input vector $X_{test}$ is classified by assigning it to the class associated with the polynomial that has the largest output caused by that vector.

### 2.2.3 Error Criterion

Two different error functions were used for the high order polynomial network algorithm. One is the traditional regularity criterion [9] based on a mean square error measure made on a left out evaluation set, and the other is the Predicted Square Error (PSE) criterion based on an information theoretic measure [1, 2].
The *regularity criterion* is derived from a simple 1-fold cross validation training and evaluation process. The computation of this error function requires that the training set observations be divided into two mutually exclusive subsets: an estimation subset, $S_{ee}$, and an evaluation subset, $S_{ev}$. $S_{ee}$ is used to determine the regression equation coefficients, while $S_{ev}$ is used to calculate the regularity criterion error, $r_k^2$.

$$r_k^2 = \frac{\sum_{p \in S_{ee}} (d(p) - z_k(p))^2}{\sum_{p \in S_{ee}} d^2(p)}, \quad k = 1, \ldots, Q, \quad \frac{1}{2} Q(Q - 1) + Q$$ (2.23)

where $d(p)$ is the desired output with input vector $X(p)$, and $z_k(p)$ is the newly derived $p^{th}$ observation resulting from the $k^{th}$ polynomial function. Note that $r_k^2$ is computed for each polynomial. From 2.23, the *regularity criterion* can be viewed as a *root mean squared* (rms) error between the actual output of the polynomial and the desired one/zero output.

There are a couple of disadvantages with using this error criterion. One is that because of the partitioning of the training set into $S_{ee}$ and $S_{ev}$ subsets, there are fewer training patterns with which to estimate the polynomial coefficients. Another is that repeatedly checking on one set of data can lead to overfitting of the polynomial model to the particular evaluation set. In effect, the evaluation subset is not fulfilling its intended purpose: to provide an independent measure of the performance of the model on unseen data.

The *predicted squared error* criterion tries to avoid partitioning the training set by defining an error function that can allow the entire training set to be used both in estimating the polynomial coefficients and in deriving a useful error measure. The PSE measure for polynomial $k$ is defined as:

$$PSE_k = \frac{1}{N} \sum_{p=1}^{N} [d(p) - z_k(p)]^2 + \frac{2\sigma_p^2 W}{N}$$ (2.24)

where $\sigma_p^2$ is the *estimate* of the error variance of the optimum, but unknown, model, $N$ is the total number of training vectors, and $W$ is the number of estimated coefficients in polynomial $k$. $W$ takes into account all the coefficients in the polynomials from the first layer to the current layer that contribute to polynomial $k$.

The PSE is the sum of two distinct terms. One is the *mean squared error* on the training set between the outputs of the polynomial and the desired outputs of 0's and
1's. The other is a *penalty term* that penalizes overly complex models. Complexity here is measured by the number of coefficients that are in the entire polynomial. More training patterns can balance out the effect of the number of coefficients, meaning that complex models are allowed only if there is enough training data.

For a fixed training set size, $N$, the mean square error curve, as a function of the complexity of the model, typically looks like the dashed curve in Figure 2.7. The penalty term (dotted curve in Figure 2.7) is a linear function of the complexity of the model, $W$, whose slope is the ratio of the error variance and the number of training vectors: $\sigma^2_p/N$. The sum of these two curves is the predicted square error which is shown in Figure 2.7 as the solid curve.

Note that the PSE curve has a minimum value which corresponds to the point (complexity) at which the GMDH algorithm stops the growth of the network. This minimum point on the PSE curve can be moved by changing the slope of the penalty term. Having a large amount of training data, ie. a large $N$, will cause the slope of the penalty term curve to be small. This will push the PSE minimum further to the right, thereby allowing a more complex model. Similarly, having little training
data will restrict the model to be relatively simple. Adjusting the variance term, $\sigma^2_\beta$, will also shift the minimum. A large variance will force a simpler model, while a small variance will allow a more complex model. Barron [2] has shown that this PSE criterion is superior to the regularity condition on a number of problems.

One criticism about the use of the PSE criterion is that it requires a prior estimate, $\sigma^2_\rho$, of the error variance, which may be difficult to determine. Barron [2] points out that there is a relatively simple estimate that upper-bounds the error variance and does not depend on the model considered. He suggests using the variation of the dependent variable $d$, given by

$$
\sigma_d^2 = \frac{1}{N} \sum_{p=1}^{N} (d(p) - \overline{d})^2, \text{ where } \overline{d} = \frac{1}{N} \sum_{p=1}^{N} d(p).
$$

This estimate is useful only if $d$ is a good estimate of the variation of the data. For classification problems, where $d$ is either 0 or 1 depending on the class of the data vector, $d$ depends only on the prior probabilities of the classes and not on the class distributions. Take for example two different two-class distributions, $\mathcal{D}_1$ and $\mathcal{D}_2$. Both have 100 vectors for class A and 100 vectors for class B, but $\mathcal{D}_1$ has a unimodal distribution for class A, while $\mathcal{D}_2$ has a more complex 10-modal distribution for class A. Using the estimate given in 2.25, both of these distributions will have the same $\sigma_d^2$, namely 0.25, which is not reflective of the true variation of the data. In Section 4.2, it is shown that a nearest neighbor classifier can be used to obtain a better measure of $\sigma_d^2$.

2.3 Linear Decision Tree Classifier

The linear decision tree classifier constructs a binary tree from the training data by repeatedly splitting the data with hyperplanes until all the vectors in the training set are separated correctly according to their class labels. Figure 2.8 shows a simple linear decision tree. The circular nodes represent non-terminal nodes, while the square nodes represent terminal nodes. Each non-terminal node in the tree splits the data coming into it into two subsets with a hyperplane that results in the “best separation” of the data. “Best separation” occurs when the hyperplane manages to separate all
classes completely to either side of the split and separates as few same class vectors as possible. The splitting criteria (i.e. the slope and offset of the hyperplane) are determined using a heuristic searching algorithm described in chapter 5 of Breiman et al's book, Classification And Regression Trees (CART) [5]. Each terminal node of the tree has associated with it a class label which identifies all vectors that fall within its bounds as belonging to that class. The top node in the tree is known as the root node.

The main difference between a linear decision tree and a binary decision tree [5, 12] is the splitting rule associated with each non-terminal node. In the binary decision tree, the splits are made on only one variable (input dimension) at a time. This restricts all the hyperplane splits to be perpendicular to the coordinate axes. The linear decision tree generalizes the hyperplane splits by allowing hyperplanes of arbitrary orientation (i.e. slope and offset). This should presumably provide more flexibility in forming decision regions.

Classification in the linear decision tree is performed by successively applying the splitting rule at each non-terminal node through which the unknown vector passes,
starting at the root node, until the vector falls into a terminal node. It is then
classified as belonging to the class associated with that terminal node.

There are many ways to construct a decision tree given a fixed set of training data.
One method is to build up the tree node by node, stopping when some “stopping
criteria” is met. Another method is to grow the tree to its maximal size (ie. until all
training patterns are correctly classified), and then strategically prune away nodes
that are not efficient in forming the classification decisions. Associated with the
pruning process is a strength measurement for each node. Empirical studies have
shown that even complex stopping rules are not effective in controlling the growth of
a decision tree [5]. As a result, the two stage method of building an overly complex
tree and then pruning away the inefficient nodes was used to construct the linear
decision trees in this study.

2.3.1 Training Algorithm

Growing the Tree

A detailed algorithmic description of the linear tree classifier used in this study is
shown in Figures 2.9 and 2.10. Initially, the vectors in the training set, \( X(p), p = 1, \ldots, N \), along with their associated class labels, \( C(p) \) are stored as exemplars in a
set called \( S \). A root node, \( t_o \), for the linear decision tree, which contains the entire
training set \( S \), is then created.

The next step is to grow the tree by repeatedly adding more nodes, each splitting
the training data with a hyperplane, until all the vectors in the training set are
separated correctly according to class. The hyperplane split, \( s \), associated with each
non-terminal node of the linear decision tree is formed from a linear combination of
the input variables or dimensions:

\[
s \equiv \sum_{i=1}^{D} a_i x_i \leq c \tag{2.26}
\]

where \( x_i \) is an element of the input vector \( X = [x_1, x_2, \ldots, x_D] \), \( a_i \) is the coefficient
or weight associated with element \( x_i \), and \( c \) is a scalar value.
1. Initialization

   (a) Store all training patterns and class labels as exemplars in a set labeled $S$.
   (b) Create a root node, $t_o \equiv t^*$, for the tree.

2. Build the Decision Tree

   (a) Compute the misclassification rate of node $t^*$
   \[
   r(t^*) = 1 - \max_j [P(j|t^*)], \quad j = 1, \ldots, M,
   \]
   \[
   P(j|t^*) = \frac{\text{total number of class } j \text{ patterns at } t^*}{\text{total number of patterns at } t^*}.
   \]
   (b) If $r(t^*) < \tau$, for some stopping criteria $\tau$, then goto 2g.
   (c) Find the Optimal Splitting Criteria, $s^*$, at node $t^*$, where
   $s^*$ is the split corresponding to $\Phi(s^*, t^*) = \max_i [\Phi(s_i, t^*)]$
   \[
   \Phi(s_i, t^*) = 2P_L P_R \sum_{j=1}^M [P(j|t_L) - P(j|t_R)],
   \]
   where
   \[
   t_{L,R} \equiv \text{left, right offspring of node } t, \quad P_{L,R} = \frac{\text{total number of patterns at } t_{L,R}}{\text{total number of patterns at } t}.
   \]
   (d) $s^*$ can be expressed as the rule: Is $\sum_{i=1}^D a_i x_i \leq c$?
   (e) $\forall x_i \in S$, split the set $S$ into $S_L$ and $S_R$ according to:
   \[
   x_i \in \begin{cases} 
   S_L & \text{if } \sum_{i=1}^n a_i x_i \leq c \\
   S_R & \text{otherwise} 
   \end{cases}
   \]
   (f) Associate $t_L$ to $S_L$ and $t_R$ to $S_R$.
   (g) Let $t^*$ be a node not yet split (ie. $t_L$ or $t_R$).
   (h) Set $S$ to be the set of patterns associated with $t^*$.
   (i) Split $t^*$ by going to step 2.

3. Prune the Decision Tree

   (a) Compute the strength of each node $t$: $g(t) = \frac{R(t)}{|T_t| - 1}$, where
   \[
   R(t) = r(t) p(t) = r(t) \frac{\text{number of patterns at node } t}{\text{total number of patterns in training set}},
   \]
   \[
   R(T_t) = \sum_{t' \in T_t} R(t'),
   \]
   and $|T_t|$ is the number of terminal nodes in the subtree headed by node $t$.
   (b) Prune away the subtree headed by the weakest node $t^w$.
   (c) Convert node $t^w$ to a terminal node.
   (d) Recompute $g(t)$ of the ancestors of $t^w$.
   (e) Prune tree to an "acceptable" size by repeating steps 3a - 3d.

---

Figure 2.9: The Linear Decision Tree Classifier Training Algorithm.
1. Present Input Pattern $X_{\text{test}}$ to the Classifier.

2. Classify $X_{\text{test}}$
   
   (a) Determine which terminal node $X_{\text{test}}$ falls into by successively applying the splitting rule $s^*$ at each non-terminal node.
   
   (b) Classify $X_{\text{test}}$ as belonging to the class associated with the terminal node into which it falls.

Figure 2.10: The Linear Decision Tree Classifier Classification Algorithm.

This splitting rule partitions the data coming into the node into two parts, namely those that satisfy equation 2.26, $S_L$, and those that don't, $S_R$. Successive partitionings of this form generate decision regions that are complex hyper-polygons. Figure 2.11 shows the decision regions formed for a ten-class, two-dimensional problem. The decision boundaries are straight lines that combine to form arbitrarily complex decision regions.

Growing the tree essentially involves determining the set of coefficients, $\{a_i\}$, and the offset, $c$, for each non-terminal node that results in the "best split" of the data coming into that node. The criterion used to select the "best split", $s^*$, of a non-terminal node, $t$, is

$$\Phi(s^*, t) = max_i [\Phi(s_i, t)].$$

(2.27)

The "goodness" of a split $s_i$ at node $t$, is defined as:

$$\Phi(s_i, t) \equiv 2P_LP_R \sum_{j=1}^{M} |P(j|t_L) - P(j|t_R)|$$

(2.28)
Figure 2.11: Decision Regions formed by the Linear Tree Classifier on the 10 Class, 2-Dimensional Vowel Problem.
where $M$ is the number of classes, and

$$
\begin{align*}
    t_L &\equiv \text{left offspring of node } t \\
    t_R &\equiv \text{right offspring of node } t \\
    P_L &= \frac{\text{total number of patterns at } t_L}{\text{total number of patterns at } t} \\
    P_R &= \frac{\text{total number of patterns at } t_R}{\text{total number of patterns at } t} \\
    P(j|t_L) &= \frac{\text{total number of class } j \text{ patterns at } t_L}{\text{total number of patterns at } t_L} \\
    P(j|t_R) &= \frac{\text{total number of class } j \text{ patterns at } t_R}{\text{total number of patterns at } t_R}.
\end{align*}
$$

Note that $\sum_{j=1}^{M} P(j|t) = 1$ and $P_L + P_R = 1$. According to equation 2.28, $\Phi(s_i, t)$ is maximized when a split, $s_i$, manages to separate all classes completely to either offspring of the node $t$. In this case, each $|P(j|t_L) - P(j|t_R)|$ for $j = 1, \ldots, M$ will yield a maximum value of 1. However, if the split tends to separate vectors of the same class, then each $|P(j|t_L) - P(j|t_R)|$ and the resulting $\Phi(s_i, t)$ will be small.

The process of finding local optimum splitting rules will stop when all input vectors at a node belong to a single class. The node then becomes a terminal node bearing the label of that class. If the training set is large, or if memory constraints are severe, a misclassification rate can be used to halt the growth of a tree. The misclassification rate of a node, $t$, is defined as:

$$
r(t) = 1 - \max_j [P(j|t)], j = 1, \ldots, M. \quad (2.29)
$$

When a node, $t$, contains only patterns of a single class, $r(t)$ is 0. Thus, a simple rule to halt the growth of a tree is to stop splitting at a node, $t$, if $r(t) < \tau$, where $0 < \tau < 1$ is some threshold.

**Searching for the Best Linear Combination**

The concept of searching a high dimensional space for the set, $\{a_i, c\}$, corresponding to the “best split”, $s^*$, that maximizes $\Phi(s_i, t)$ is easily stated, but the implementation in terms of an effective search algorithm is complex. Breiman [5] gives a heuristic
search algorithm, described below, for finding the set of \( \{a_i, c\} \) used in the non-terminal nodes. Alternative approaches include using a variant of the perceptron convergence procedure, a mean square error as with the GMDH network, or principal component analysis. The CART algorithm is not guaranteed to find the absolute "best split" \( s^* \). Like other search algorithms, it may get trapped in local maxima.

Let \( X(p), p = 1, \ldots, N \) be the set of training vectors at node \( t \), where each \( X(p) \) is made up of \( D \) components: \( X(p) = [x_1(p), x_2(p), \ldots, x_D(p)] \). At the first step of the algorithm, each component \( i \), for \( i = 1, \ldots, D \), of the \( N \) vectors \( x_i(p) \) is centered at the component median and divided by the component interquartile range. Since only these normalized values are used, they will also be denoted by \( x_i(p) \).

The algorithm then cycles through the set of \( D \) variables \( x_1, x_2, \ldots, x_D \), doing a search for an improved linear combination split at each iteration. The initial split for the first iteration is the best univariate split, \( s_{uni}^* \), which is defined as the rule of the form

\[
x_i \leq c \tag{2.30}
\]

that maximizes \( \Phi(s_{uni}, t) \). The total number of possible unique rules of the form 2.30 at node \( t \) is \( n \cdot N \), meaning that an exhaustive search can be used to find the best \( \{x_i, c\} \) set, \( \{x_i^*, c^*\} \).

At the beginning of the first iteration, let the current linear combination split be

\[
v_1 \equiv \sum_{i=1}^{D} \beta_i x_i \leq c_1. \tag{2.31}
\]

Since the initial split is univariate, all the \( \beta \)'s will be zero except for \( \beta_{i^*} \), which will be one. The scalar \( c_1 \) will be \( c^* \).

For a fixed \( \gamma_1 \), consider finding the best split of the form

\[
v_1 - \delta_1(x_1 + \gamma_1) \leq c_1, \tag{2.32}
\]

where \( \delta_1 \) ranges over all possible values. This is done by rearranging 2.32 as

\[
\delta_1(x_1 + \gamma_1) \geq v_1 - c_1 \tag{2.33}
\]
so that

\[ \delta_1 \geq \frac{v_1 - c_1}{x_1 + \gamma_1}, \text{ for } x_1 + \gamma_1 \geq 0 \]
\[ \delta_1 \leq \frac{v_1 - c_1}{x_1 + \gamma_1}, \text{ for } x_1 + \gamma_1 < 0 \]

Now let

\[ u(p) = \frac{v_1(p) - c_1}{x_1(p) + \gamma_1}, \quad p = 1, \ldots, N. \tag{2.34} \]

Order the \( N \) values of \( u(p) \), and find the best split of the form

\[ u \leq \delta_1^A \quad \forall \ u(p) \text{ such that } x_1(p) + \gamma_1 \geq 0 \]
\[ u \geq \delta_1^B \quad \forall \ u(p) \text{ such that } x_1(p) + \gamma_1 < 0 \]

as \( \delta_1^A \) and \( \delta_1^B \) range over all values. Note that there are only \( N \) distinct values of \( \delta_1^A \) and \( \delta_1^B \). Take the best of these two splits and let \( \delta_1 \) be the corresponding threshold.

This procedure is carried out for three values of \( \gamma_1 \): \( \gamma_1 = 0 \), \( \gamma_1 = -\frac{1}{4} \), and \( \gamma_1 = \frac{1}{4} \). For each value of \( \gamma_1 \), the best split of the form 2.32 is found. The three resulting splits are then compared and the best \( \delta_1 \) and \( \gamma_1 \), are used to update \( v_1 \):

\[ v_2 = \sum_{i=1}^{D} \beta_i' x_i, \tag{2.35} \]

where

\[ \beta_1' = \beta_1 - \delta_1 \tag{2.36} \]
\[ \beta_i' = \beta_i, \quad i > 1 \tag{2.37} \]

and

\[ c_2 = c_1 + \delta_1 \gamma_1. \tag{2.38} \]

Note that since only the first component variable, \( x_1 \), has been examined, only one weight coefficient, namely \( \beta_1 \), has been determined. In order to determine \( \beta_2 \), the above procedure is repeated using variable \( x_2 \) and the updated \( v_2 \). In other words, a search is made for the best split of the form:

\[ v_2 + \delta_2(x_2 + \gamma_2) \leq c_2, \quad \text{where } \gamma_2 = 0, -\frac{1}{4}, \frac{1}{4}. \tag{2.39} \]
This is repeated for variables $x_3, x_4$, and so on, until variable $x_D$ is used. At this point, the updated split is of the form $v_D \leq c_D$, and all the $\beta_i$’s have been determined. The first iteration is then completed by finding the best split of the form $v_D \leq c'_D$ as $c'_D$ ranges over all values. This search is essentially a “univariate” search where $c'_D$ has only $N$ unique values corresponding to the $N$ values that $v_D$ can take on. This split forms the initial split for the next (second) iteration.

Let $s_L$ be the linear combination split produced at the end of the $L^{th}$ iteration. The cycling terminates if

$$|\Phi(s_L, t) - \Phi(s_{L-1}, t)| \leq \epsilon,$$

where $\epsilon$ is a small threshold. After the final linear combination split is determined, it is converted to the split in 2.26 on the original non-normalized variables, by adding back the median value and multiplying by the variance of each component.

**Pruning the Tree**

The misclassification rate given in 2.29 can be viewed as an “impurity” measurement of a node. A node $t$ with zero impurity means that all patterns that end up in that node belong to only one class, corresponding to a misclassification rate $r(t) = 0$. Using this idea, a measurement of strength, $g(t)$, of a node $t$ can be expressed as

$$g(t) = \frac{R(t) - R(T_i)}{|T'_i| - 1},$$

where

$$R(t) = r(t)p(t)$$

$$= r(t) \frac{\text{number of patterns at node } t}{\text{total number of patterns in training set}}$$

$$R(T_i) = \sum_{t' \in T'_i} R(t')$$

and $T'_i$ is the set of terminal nodes in the subtree headed by node $t$, and $|T'_i|$ is the number of terminal nodes in that subtree.

$R(t)$ can be viewed as a normalized misclassification rate of node $t$. The strength $g(t)$ of node $t$ is highest when $R(t)$ is much greater than the misclassification rate
of its subtree, \( R(T_1) \). This means that \( g(t) \) is large for nodes whose subtrees are useful in performing classification, and small for nodes whose subtrees are not useful in performing classification. Thus, \( g(t) \) provides a measure of the "classification efficiency" of a subtree headed by node \( t \). In addition, the strength measurement also takes into account the magnitude of \( T'_t \) to incorporate the cost of maintaining nodes.

This strength measurement can be used to prune away nodes that are not efficient in forming classification decisions after a tree is completely grown. The strength measurement is first computed for all non-terminal nodes in a tree. The subtree of node \( t^* \) with the smallest \( g(t^*) \) is then pruned from the original tree. Without any offspring, node \( t^* \) is converted to a terminal node of the new tree. If further pruning of the tree is desired, the strength of all ancestors of \( t^* \) would have to be recomputed before finding and deleting the next node.

## 2.4 Gaussian Mixture Classifier

Bayesian classifiers explicitly compute and utilize the statistical information contained in the distributions of the input data to perform classification. Given a \( D \)-dimensional input vector \( \mathbf{X} \), the goal of a Bayesian classifier is to select the one class, \( C \), out of the \( M \) possible classes, that satisfies

\[
\text{class} = C = \text{ArgMax}_C \left[ P(C|X) \right] \tag{2.42}
\]

where \( P(C|X) \) is the conditional or Bayes \textit{a posteriori} probability of class \( C \) given input \( X \). This conditional probability can be estimated explicitly using \textit{Bayes Theorem} by rewriting it as

\[
P(C|X) = \frac{p(X|C)P(C)}{p(X)} \tag{2.43}
\]

where \( p(X|C) \) is the conditional probability density function (pdf) or \textit{likelihood} of \( X \) given that it belongs to class \( C \), and \( P(C) \) is the \textit{a priori} probability of being in class \( C \). The denominator \( p(X) \) is the combined probability density function (pdf) of the input \( X \) across all classes. Since \( p(X) \) is common to all \( P(C|X) \) expressions,
regardless of the class, it can be dropped from the equations. Thus equation 2.42 can be rewritten in the following way.

\[
\text{class} = C \quad = \quad \text{ArgMax}_c \left[ P(C|X) \right] \quad \quad (2.44) \\
= \quad \text{ArgMax}_c \left[ \frac{p(X|C)P(C)}{p(X)} \right] \quad \quad (2.45) \\
= \quad \text{ArgMax}_c \left[ p(X|C)P(C) \right] \quad \quad (2.46)
\]

If the input distributions, \( p(X|C) \), for all \( M \) classes are known and fixed, a Bayesian classifier is guaranteed to achieve the optimal minimum error classification accuracy [8].

In most practical cases, however, the class distributions are not known and, as a result, have to be estimated. For a variety of reasons, including mathematical simplicity and empirical observations [4], most Bayesian classifiers perform this estimation by first assuming that \( p(X|C) \) is multivariate normal or Gaussian distributed, with certain unknown parameters, usually the mean vector and covariance matrix. Given this assumption of the underlying distributions, the estimation process reduces to the relatively straightforward task of determining appropriate values for the unknown parameters.

The Gaussian mixture classifier is a generalization of the simple unimodal Gaussian classifier. Instead of modeling the distribution of a class with just one Gaussian distribution, a weighted sum, or mixture of Gaussian distributions is used [8]. Figure 2.12 shows a two class distribution modeled with a linear combination of Gaussian distributions. In this example, there are two Gaussian components for each class model. An obvious advantage of using multiple Gaussians is that more general and complex distributions can be modeled. These include multi-modal class distributions and high-tailed distributions. The unknown parameters, which include the means, covariances, and weights, of the Gaussian components for each class are estimated from the training data using a maximum likelihood approach and the iterative Estimate-Maximize or E-M algorithm [8, 15].
2.4. GAUSSIAN MIXTURE CLASSIFIER

Figure 2.12: Two Class Gaussian Mixture Distribution.

2.4.1 Training Algorithm

A detailed algorithmic description of the Gaussian mixture classifier used in this study is shown in Figures 2.13 and 2.14. As mentioned above, the Gaussian mixture classifier assumes that the underlying class distribution, $p(X|C)$, for each class is a weighted sum of Gaussians:

$$p(X|\theta_s, C) = \sum_{i=1}^{G} \pi_i p(X|\theta_i, C)$$

(2.47)

where

$$p(X|\theta_i, C) = \frac{1}{(2\pi)^{d/2}|C_i|^{1/2}} \exp \left[ -\frac{1}{2}(X - \mu_i)^T C_i^{-1}(X - \mu_i) \right].$$

(2.48)

In these equations, $\theta_i = \{\mu_i, C_i\}$ corresponds to the unknown mean, $\mu_i$, and covariance, $C_i$, parameters, $G$ is the number of mixture components, and $\pi_i$, $i = 1, \ldots, G$ are the mixing parameters or component weights.

Once the number of mixture components, $G$, is fixed, the unknown parameters $\mu_i$, $C_i$, and $\pi_i$ of the pdf $p(X|\theta_s, C)$ for each class can then be estimated iteratively using the Estimate-Maximize (E-M) algorithm. To start the first iteration of the algorithm, initial estimates of these parameters have to be provided. In the algorithm described
1. Initialization

(a) Select one of the $M$ classes, $C$, in the problem to be modeled.
(b) Input training patterns, $X_p$, $p = 1, \ldots, N_c$ from class $C$ to the classifier.
(c) Model the underlying class $C$ distribution as a weighted sum of Gaussians:

$$p(X|\theta, C) = \sum_{i=1}^{G} \pi_i p(X|\theta_i, C)$$

where $\theta_i = \{\mu_i, C_i\}$, $G$ is the number of mixture components,

$$p(X|\theta_i, C) = \frac{1}{(2\pi)^{D/2}|C_i|^{1/2}} \exp \left[ -\frac{1}{2}(X - \mu_i)^T C_i^{-1} (X - \mu_i) \right],$$

and

$$\pi_i, i = 1, \ldots, G$$

are the mixing parameters or component weights.
(d) Fix the number of component mixtures, $G$.
(e) Initialize estimates of the component means, $\mu_i$, $i = 1, \ldots, G$ to random training samples.
(f) Initialize estimates of the covariance matrices, $C_i$, to the Identity matrix, $I$.
(g) Initialize $\pi_i = \frac{1}{G}$, so all components are equally likely.
(h) Define $\tau_{ip} \equiv P(G_i|X_p) = \frac{\pi_i p(X_p|\theta_i, C)}{p(X_p)} = \frac{\pi_i p(X_p|\theta_i, C)}{\sum_{j=1}^{G} \pi_j p(X_p|\theta_j, C)}$

(i) Compute $\tau_{ip}$ using the initial estimates of $\pi_i$, $\mu_i$, and $C_i$.

2. Iteratively Update Means, Covariances, and Weights

(a) Update weights: $\pi_i(t+1) = \frac{1}{N_c} \sum_{p=1}^{N_c} \tau_{ip}(t)$.
(b) Update means: $\mu_i(t+1) = \frac{1}{N_c \pi_i(t)} \sum_{p=1}^{N_c} \tau_{ip}(t) X_p$.
(c) Update covariances: $C_i(t+1) = \frac{1}{N_c \pi_i(t)} \sum_{p=1}^{N_c} \tau_{ip}(t) (X_p X_p^T - \mu_i(t) \mu_i^T(t))$.
(d) Recompute $\tau_{ip}$ using the new $\pi_i(t+1)$, $\mu_i(t+1)$, and $C_i(t+1)$ estimates.
(e) If $\Delta \tau_{ip} \equiv \tau_{ip}(t+1) - \tau_{ip}(t) > threshold$, then goto 2a. Otherwise, stop.

3. Repeat Steps 1 and 2 for all $M$ classes

---

Figure 2.13: The Gaussian Mixture Classifier Training Algorithm.
1. Present Input Pattern $X_{test}$ to the Classifier

2. Classify $X_{test}$ according to: $\text{class} = C = \text{ArgMax}_C [P(C)p(X_{test}|\theta_s, C)]$
   where $P(C) = N_c/N$ is the a priori probability of class $C$ estimated by counting the number of training patterns.

Figure 2.14: The Gaussian Mixture Classifier Classification Algorithm.

in Figure 2.13, estimates of $\mu_i$ are initialized to random samples from the training set, estimates of $C_i$ are set to the identity matrix $I$, and the component weights, $\pi_i$, are set to be all the same at $1/N$.  

With these initial estimates, $\tau_{ip}$, the conditional probability of Gaussian component $i$ given input $X_p$,

$$\tau_{ip} \equiv P(G_i|X_p) = \frac{\pi_i p(X_p|\theta_i, C)}{p(X_p)} = \frac{\pi_i p(X_p|\theta_i, C)}{\sum_{j=1}^{G} \pi_j p(X_p|\theta_j, C)} \tag{2.49}$$

is computed.

This value of $\tau_{ip}$ is then used in the following three update equations:

$$\pi_i(t + 1) = \frac{1}{N_c} \sum_{p=1}^{N_c} \tau_{ip}(t) \tag{2.50}$$

$$\mu_i(t + 1) = \frac{1}{N_c \pi_i(t)} \sum_{p=1}^{N_c} \tau_{ip}(t) X_p \tag{2.51}$$

$$C_i(t + 1) = \frac{1}{N_c \pi_i(t)} \sum_{p=1}^{N_c} \tau_{ip}(t) (X_p X_p^T - \mu_i(t)\mu_i^T(t)) \tag{2.52}$$

where $N_c$ is the number of training vectors, $X_p$, in class $C$. These new estimates of $\mu_i$, $C_i$, and $\pi_i$ are then used to compute a new updated value of $\tau_{ip}$. This iteration proceeds until $\tau_{ip}$ converges:

$$\Delta \tau_{ip} \equiv \tau_{ip}(t + 1) - \tau_{ip}(t) < \text{threshold}. \tag{2.53}$$

This estimation procedure is repeated for each of the $M$ different classes to obtain $M$ class distributions, $p(X|C)$. An unknown input vector, $X_{test}$ is then classified by assigning it to the most likely class as defined in equation 2.46.
If each dimension of the input vector is statistically independent, then the covariance matrix will become a diagonal matrix with the variance of each dimension on the diagonal. Furthermore, if the variances of all the component distributions, $G_i$, within each class are the same, then there will only be one within class grand covariance matrix, $C_i$, for each class $C$. If the variances of all the component distributions from all the different classes are the same, then there would be only one across class grand covariance matrix, $C_i$, for all the component distributions. In addition, each class can use the same (common) set of Gaussian components to estimate its distribution. In this case, the means and covariances of the Gaussian components will be the same for each class or tied across classes, and the only difference will be in the set of component weights. These special cases are often used to estimate the full covariance matrix because they reduce the number of parameters that have to be estimated. Sometimes, these approximations can reduce the amount of computation drastically without substantially increasing the error rate. In fact, if there is very little training data, a simplified covariance matrix can lead to better performance than a full covariance matrix. The effect of various covariance matrix estimates on the performance of the Gaussian mixture classifier is explored in Section 4.4.

It is interesting to note the similarity between a tied Gaussian mixture classifier and a radial basis function classifier. In both cases, the classifiers make use of a linear combination of a common set of Gaussians to try to classify vectors in the feature space. The main difference between the two is in what each is trained to do. The RBF classifier is trained to maximize the discrimination of one class from all the rest, while the Gaussian mixture classifier is trained to model each of the class distributions separately.

### 2.5 K-Nearest Neighbor Classifiers

A nearest neighbor classifier stores all training patterns and class labels as exemplar patterns in a database to be used during classification. Any new pattern presented to this classifier is classified to be in the same class as the nearest stored exemplar in the database. The distance metric used to determine how close two patterns are
is usually the Euclidean distance between the two patterns. A *K nearest neighbor* classifier differs from a nearest neighbor classifier only in that the classification is now based not just on the nearest neighbor, but on the *K* nearest neighbors. In this case, the classification is determined using a voting process by simply choosing the class with the most representatives in the *K* selected exemplars. Any ties are broken randomly.

The K-nearest neighbor (KNN) classifier is simple, very easy to implement, and requires no training. It does, however, require the storage of all the training data, and the computation of many distances to search for the nearest neighbors during classification. For large training sets, this results in large training and classification memory requirements, and slow classification times.

There are many ways to improve the efficiency of the KNN classifier. The *kd-tree* implementation of the KNN, discussed below, tries to build some structure into the stored database and the search procedure in order to reduce the number of distance calculations required during classification. Other algorithms, like the *condensed k-nearest neighbor* classifier, store only key patterns from the training set, thereby reducing the memory requirements and the number of distance calculations.

### 2.5.1 Condensed K-Nearest Neighbor Classifier

The condensed KNN classifier is sequentially trained by applying training patterns individually and adding a new pattern to the exemplar database only when that pattern is misclassified using the exemplars stored so far. The goal of this technique is to reduce the total number of stored exemplars while keeping the ones that are important in forming decision boundaries [8]. A detailed algorithmic description for the condensed nearest neighbor classifier is shown in Figures 2.15 and 2.16.

Initially, *K'* training patterns are stored as the initial exemplars of the classifier. The classifier database is then iteratively constructed by examining successive training patterns and classifying them by assigning them to the class with the most representatives in the nearest *K'* neighbors in the stored database. If an input pattern is *incorrectly* classified, it is then added as an exemplar to the database. If it is *correctly* classified, however, it is discarded and not kept as an exemplar.
1. Initialization

(a) Determine $K'$, the number of nearest neighbors to use in classifying the training patterns.

(b) Input $K'$ training patterns, $X(p), p = 1, \ldots, K'$, to the classifier and store them as exemplar patterns along with their class labels $C(p)$.

2. Iteratively Build Classifier

(a) Input another training pattern, $X(p)$, to the Classifier.

(b) Compute Euclidean distances from $X(p)$ to all exemplars stored so far.

(c) Select the $K'$ nearest patterns to $X(p)$.

(d) Classify $X(p)$ as belonging to the class with the most representatives in the set of $K'$ selected nearest neighbors.

(e) If $X(p)$ is misclassified, store $X(p)$ as another exemplar along with the class $C(p)$, Otherwise discard it.

(f) If there are more training patterns, goto 2a.

Figure 2.15: The Condensed K-Nearest Neighbor Classifier Training Algorithm.

1. Present Input Pattern $X_{test}$ to the Classifier

2. Search for K Nearest Neighbors

(a) Compute Euclidean distances from $X_{test}$ to all stored exemplars.

(b) Select the $K$ nearest patterns to $X_{test}$.

3. Voting and Classification Procedure

(a) Classify $X_{test}$ as belonging to the class with the most representatives in the set of $K$ selected nearest neighbors.

(b) Break ties randomly.

Figure 2.16: The Condensed K-Nearest Neighbor Classifier Classification Algorithm.
This classification/storage process continues until all the training patterns have been examined. An unknown input vector, \( X_{\text{test}} \), is then classified by finding the \( K \) nearest neighbors to \( X_{\text{test}} \) and assigning it to the class with the most representatives in the selected set of exemplars.

Since this algorithm is trained sequentially, looking at only one training pattern at a time, it is sensitive to the ordering of the patterns in the training set. One way to get around this problem is to train the classifier by cycling through the training patterns multiple times until no new patterns are added to the stored exemplar database. Another method is to run the algorithm a number of times on different permutations of the training set, and then take the union of the stored exemplars across the different trials and run it through the algorithm one more time to obtain a set of exemplar vectors for the database. Both of these different training methods were explored and are described in section 4.5.

Figure 2.17 shows a stylized two dimensional feature space with two different classes, namely A and B, and the decision boundary (solid line) between the two classes. The condensed nearest neighbor algorithm attempts to keep training vectors...
1. Initialization

(a) Store all training patterns and class labels as exemplars in a set labeled $S$.
(b) Create a root node, $t_0 \equiv t^*$, for the tree.

2. Recursively Build the Tree

(a) Compute the variance, $\sigma^2$, of each input dimension, $k$, using all $X(p) \in S$.
(b) Select the input dimension, $k_o$, with the largest $\sigma^2$ and compute its median, $\mu_o$.
(c) Associate the pair ($k_o, \mu_o$) with the node $t^*$.
(d) $\forall X(p) \in S$, split the set $S$ into $S_L$ and $S_R$ according to:
   
   $X(p) \in \begin{cases} 
   S_L \text{ if the } k_o^{th} \text{ element } \leq \mu_o \\
   S_R \text{ otherwise}
   \end{cases}$

(e) Associate $t_L$ to $S_L$ and $t_R$ to $S_R$.

(f) If the number of vectors, $X(p)$ in $S_{x \in \{L,R\}}$ is less than the bucket size, then DONE, Otherwise set $S \equiv S_{x \in \{L,R\}}$ and GOTO 2a.

Figure 2.18: The KD Tree (KNN) Classifier Training Algorithm.

that are important in forming the required decision boundaries. This translates to
keeping more of the vectors (the bold A's) near the actual decision boundary and fewer
of the vectors (the small A's) in the interior of the decision region (see Figure 4.24 in
Section 4.5).

2.5.2 KD Tree Classifier

A KD tree classifier is a tree-based implementation of a KNN classifier which attempts
to reduce classification time on a serial computer by building structure into the stored
database and the search procedure [10, 18].

A detailed algorithmic description of the KD tree algorithm used in this study
is given in Figures 2.18 and 2.19. A binary splitting tree, very much like the linear
decision tree, is constructed from the training data by repeatedly splitting the feature
space with hyperplanes until the space is divided up into small hyper-rectangular
2.5. **K-NEAREST NEIGHBOR CLASSIFIERS**

---

1. Present Input Pattern $X_{test}$ to the Classifier

2. Recursively Search for the K Nearest Neighbors
   
   (a) Initialize a list, $L_K$, of the K nearest neighbors and their distances.
   
   (b) Start recursion by setting current node $t^*$ equal to the root node, $t_o$.
   
   (c) If $t^*$ is a terminal node, then
      
      i. Compute all distances from $X_{test}$ to all vectors in the bucket, updating the list $L_K$.
      
      ii. Let the $K^{th}$ nearest distance be $R$. If there are less than $K$ distances, set $R = +\infty$.
      
      iii. If $R$ is within the bounds of the current bucket as defined by the $(k_o, \mu_o)$ pairs associated with all the ancestor nodes of the bucket, then DONE else RETURN to calling function.
   
   (d) If $t^*$ is a non-terminal node, then
      
      i. Using the $(k_o, \mu_o)$ pair associated with node $t^*$, if the $k_o^{th}$ element of $X_{test} \leq \mu_o$ then
         
         * set $t^* \equiv t_L$ and goto 4c.
         
         * if bounds of $t_R$ overlap $R$: set $t^* \equiv t_R$, goto 4c.

      Otherwise
         
         * set $t^* \equiv t_R$ and goto 4c.
         
         * if bounds of $t_L$ overlap $R$: set $t^* \equiv t_L$, goto 4c.
      
      ii. If $R$ is within the bounds of the current bucket, then DONE else RETURN to calling function.

3. Voting and Classification Procedure
   
   (a) Classify $X_{test}$ as belonging to the class with the majority in $L_K$.
   
   (b) Break ties randomly.

---

Figure 2.19: The KD Tree (KNN) Classifier Classification Algorithm.
regions, called \textit{buckets}, each containing only a few data vectors. These small regions are mutually exclusive, i.e. non-overlapping, and correspond to the \textit{terminal nodes}, or leaves, of the tree. The maximum number of data vectors in each bucket, or the \textit{bucket size}, can be used to specify the number of terminal nodes, and hence, the size and complexity of the tree.

Each of the \textit{non-terminal nodes} in the tree represents a \textit{subset} of the data in the training set and a hyperplane partition of that subset. The binary partition or split is made at the \textit{median of the component with the largest variance} as determined from the data at the node. The splitting criterion for each node is defined as a pair, \((k_o, \mu_o)\), where \(k_o\) is the component with the largest variance, and \(\mu_o\) is the median of that component. This splitting rule partitions the data, \(S\), coming into the node into two parts, namely those that satisfy equation 2.54, \(S_L\), and those that don't, \(S_R\).

\[
X(p) \in \begin{cases} 
S_L & \text{if the } k_o^{th} \text{ element } \leq \mu_o \\
S_R & \text{otherwise}
\end{cases} 
\tag{2.54}
\]

Thus, all the splits are \textit{perpendicular to the coordinate axis}, as in a binary decision tree. The root node of the tree represents the entire training set.

Given this partitioning of the input space, the search for the \(K\) nearest neighbors of a test vector needs only to consider a small number of \textit{buckets} in the vicinity of the test vector, and hence a very small portion of the training set. A stylized two-dimensional, two class feature space that has been divided into six buckets is shown in Figure 2.20. Let \(X\) be an unknown input vector and the number of nearest neighbors, \(K\), be three. The goal in this simple example is to search for the \(K = 3\) nearest neighbors of \(X\).

First, all of the stored vectors in the bucket into which the unknown vector, \(X\), falls must be checked. In this example, \(X\) falls into the upper-right-hand bucket. Next, a threshold radius, \(R_1\), is set to be equal to the distance from \(X\) to the \textit{third} nearest neighbor in the bucket. If there are less than three vectors in the bucket, \(R_1\) is set to \(+\infty\).

A check is then performed to see if \(R_1\) is within the \textit{bounds} of the bucket. The bounds are defined in each dimension by the accrual of the \((k_o, \mu_o)\) pairs from the
2.5. *K*-NEAREST NEIGHBOR CLASSIFIERS

![Diagram of a stylized two-dimensional two-class distribution illustrating the data vectors searched by the KD Tree KNN classifier.](image)

Figure 2.20: A Stylized Two-Dimensional Two-Class Distribution Illustrating the Data Vectors Searched by the KD Tree KNN Classifier.

ancestor nodes of the bucket. These \((k_o, \mu_o)\) pairs form successively tighter lower or upper limits on the bounds as one descends down the tree. If \(R_1\) is within the bucket, then the search is finished because no point outside the bucket will be closer to \(X\) than the 3 points already found. If \(R_1\) is not within bounds, however, then neighboring buckets will have to be searched. In this example, \(R_1\) is not within the bounds of the first bucket and neighboring buckets need to be examined.

In order to determine which neighboring buckets will have to be examined, a test is performed to see if the bounds of the neighboring buckets overlap the radius threshold \(R_1\). If the bounds overlap, this means that there may be vectors in the neighboring bucket that may be closer to \(X\) than the ones already checked, and those new vectors will have to be examined. In this example, the bucket immediately under the first bucket overlaps the radius defined by \(R_1\).

After this new bucket is searched, another radius is defined which is equal to the distance from \(X\) to the third nearest neighbor so far found. In this case, the new radius is \(R_2\). Another radius within bounds check is performed to see if the new "larger" partition, defined by the two buckets, is large enough to contain the \(K\) nearest neighbors to \(X\). In this case, these two buckets are large enough for \(K = 3\),
and the search can now stop. Note that this algorithm is guaranteed to find the nearest neighbors of \( \mathbf{X} \). For this example, to find the three nearest neighbors to an unknown input vector, \( \mathbf{X} \), only two buckets have to be searched. The other buckets in the space do not even have to be considered.

### 2.6 Chapter Summary

In this chapter, the two neural network and four conventional pattern classifiers studied in this thesis were described. Detailed algorithmic descriptions for each classifier were also given. In addition to the parameters (i.e. weights, means, etc.) that have to be estimated using training data, most classifiers have many free parameters that also have to be determined. These free parameters are, in general, difficult to estimate directly and include such things as the number of basis function nodes in the RBF classifier, the number of candidate polynomials or the width of each layer in the GMDH classifier, the number of terminal nodes in the linear tree classifier, and the number of nearest neighbors in the KD tree and condensed \( K \) nearest neighbor classifiers. Chapter 4 describes tuning experiments showing how these and other free parameters were selected for each classifier.
Chapter 3

Data Sets

In order to study the behavior and characteristics of the different classifiers on problems of varying levels of difficulty, the classifiers were trained and evaluated on four different data sets. Two sets of data, the bullseye and disjoint problems, were artificially generated. The other two sets of data, the vowel and digit problems, contained pre-processed speech data. A brief description of these four problems is presented in this chapter.

3.1 Bullseye Classification Problem

The bullseye problem consists of two dimensional vectors divided into two classes. The training set for this problem is shown in figure 3.1. Class A consists of vectors uniformly distributed within a unit circle, and class B consists of vectors uniformly distributed in an annular ring with an inner radius of 1.0 and an outer radius of 5.0. The training and test sets for this problem both have 500 patterns.
Figure 3.1: Training Set for the Bullseye Problem.
3.2  **Disjoint Classification Problem**

The disjoint problem also has *two dimensional* vectors separated into *two classes*. The training set is shown in figure 3.2. Class $A$ consists of vectors uniformly distributed in two squares, and class $B$ consists of vectors uniformly distributed around class $A$. Again, the training and test sets both have 500 patterns.
3.3 F1-F2 Vowel Recognition Problem

This problem consists of a set of two dimensional first and second formant frequency (F1-F2) vectors obtained by Peterson and Barney [20] using spectrographic analysis of vowels in words of the form: /hVd/, where /V/ is the vowel. The words were spoken by 67 people, including men, women, and children. The first and second formant frequencies of the ten different vowels were split into two sets, resulting in 398 training patterns, and 333 testing patterns. The training set for the vowel problem is shown in figure 3.3.
3.4 Digit Recognition Problem

The digit problem is made up of a set of seven monosyllabic digit patterns taken from the TI 20 Isolated Word Database. The digits are: “one,” “two,” “three,” “four,” “five,” “six,” and “eight.” A version of this database that had been sampled at 12 kHz was processed to extract 11 mel cepstra from every 10 millisecond frame. Each pattern vector consists of twenty two cepstral coefficients. Figure 3.4 illustrates how the twenty-two coefficients for each digit were obtained. One set of eleven coefficients was obtained from the highest-energy frame for the word; the other set of eleven come from the frame preceding the highest-energy frame by 30 milliseconds. The additional coefficients from the earlier frame provide information about spectral changes over time. There are 70 training patterns and 112 testing patterns for each speaker. All experiments presented here are speaker dependent, using 16 different speakers.
CHAPTER 3. DATA SETS
Chapter 4

Experimental Results

Many classifiers have free parameters that have to be adjusted in order to be able to perform well on a given classification problem. For example in the \( K \) nearest neighbor classifiers, one free parameter is the number of nearest neighbors, \( K \), to use in the voting to determine the class of an unknown vector. In the radial basis function classifiers, the width or variance of the Gaussian shaped basis functions is a free parameter that has to be set so that there is enough “overlap” between the basis functions in order to be able to form a smooth interpolation. In many cases, the determination of the optimal set of values for parameters is a trial and error process. A range of parameter values must be searched to find a suitable value.

In addition to finding suitable values for the free parameters, the complexity of the classifier must also be determined. Most classifiers exhibit a characteristic divergence between the error rates on the training set and on the testing set as a function of the complexity of the model. In general, if a classifier is trained on one set of training data, and it is allowed to grow very complex, the classifier will start to capture the idiosyncrasies of that set of data, forming decision boundaries that are too specific and that can not generalize well. As a result of this overfitting, the classifier will perform very well on the training set, but very poorly on unseen test data. Thus, it is very important to be able to determine the “proper” complexity of the classifier so that it performs well on, but at the same time avoids overfitting, the training data. This is particularly true for problems where the training set may not be a faithful representation of the unseen data in the test set. It is for this reason that good classifier models are difficult to build given a limited amount of training data.

This chapter describes the various tuning experiments performed to determine the free parameters and the proper complexity of the classifiers on the given classification tasks. For each classifier, a series of experiments was performed to determine the
effect of the free parameters on the performance of the classifier in terms of error rate. The experiments searched the space of the different parameters looking for values that result in good performance on both the training and test sets. From these experiments, an appropriate set of values for the free parameters was selected. Similar experiments were performed to determine the proper classifier size or complexity. For the GMDH and linear tree classifiers, cross validation methods on the training set were used to determine classifier complexity.

4.1 Radial Basis Function Classifier

4.1.1 Optimal Width of Basis Functions

There are only two free parameters in the RBF Classifier used in this work. One is the set of locations or mean vectors of the basis functions, and the other is the set of corresponding widths or variances of the basis functions.

Initial estimates of the widths of the basis functions are determined, as mentioned in Section 2.1, by setting the variance of each basis function, \( \sigma_i^2 \), equal to the average Euclidean distance from the basis function center to its five nearest neighbors in the training set. A global proportionality factor, \( H \), which multiplies the variance \( \sigma_i^2 \), as shown in equation 2.1, is then used to rescale these basis function widths to obtain good performance.

As discussed previously, the rationale behind setting the variance of each basis function proportional to the average distance from the basis function center to its nearest five neighbors is to have narrow basis functions in regions of dense population and wide basis functions in regions of sparse population. The reason for including the global proportionality factor \( H \) is to allow some flexibility to ensure that there is enough "overlap" in the basis functions to perform a good interpolation.

Figure 4.1 shows the error rate of the RBF classifier on the bullseye problem as a function of the proportionality factor \( H \) for a range of different numbers of basis function nodes. Regardless of the number of basis functions, as the width of the basis functions increases, the performance steadily improves, until a certain width is
Figure 4.1: Error Rate of the RBF Classifier on the Bullseye Problem as a Function of $H$ and the Number of Basis Function Nodes.
reached, after which the performance levels off, and eventually gets worse.

If the RBF classifier is viewed as a mapping interpolation, where the goal is to have the basis functions cover the space so that a smooth interpolation can be formed, then this behavior makes intuitive sense. With a very small $H$, the basis functions are very narrow and do not overlap much with the other basis functions. This makes it very difficult to perform a good interpolation and results in poor performance. As $H$ is increased, the basis functions gets wider, and begin to overlap more, resulting in better interpolation and better performance. There is a very wide range, usually a few orders of magnitude, over which the performance of the classifier stops improving and stays relatively constant. Over this range, the basis functions seem to have reached an "optimal" amount of overlap. The length of this range suggests that the RBF classifier is relatively insensitive to the actual basis function widths so long as they overlap sufficiently. As $H$ increases further and the variances get very large, all the basis functions start to look alike. In particular, all of them will have values very close to one over the entire input space of the problem. This situation, like that of the narrow basis functions, results in poor interpolation and hence poor performance.

For the bullseye problem, where there are only two classes and the decision regions are very simple, performance remains good even for very large basis function widths. A value of $H = 10$ to 100, provides good performance. On more complex problems, however, where there are more classes or more complex decision regions, narrower basis functions are required in order to form the finer grain boundaries required. This behavior can be seen in Figure 4.2, which plots the error rate of the RBF classifier on the vowel problem as a function of the proportionality factor $H$ for a range of different numbers of basis function nodes. Compared with the bullseye problem, the vowel problem, which is also two dimensional, requires much narrower basis functions in order to perform well. Above an $H$ value of 5, the performance on the vowel problem degrades rapidly. This can be attributed to the fact that the vowel problem has 10 classes, that the decision regions are very complex, and that the classes overlap.

Figure 4.3 shows the error rate of the RBF classifier on the 22-dimensional digit problem as a function of the proportionality factor $H$ for a range of different numbers of basis function nodes. Performance is best when the basis functions are wide, in
Figure 4.2: Error Rate of the RBF Classifier on the Vowel Problem as a Function of $H$ and the Number of Basis Function Nodes.
contrast to the vowel problem, where performance is best for narrow basis functions. This suggests that more smoothing is required in the high dimensional digit problem when there are very few training patterns per class.

In Figure 4.1, another noticeable characteristic is that as the number of basis functions increases, good performance can be achieved with narrower and narrower basis functions. This behavior is shown in Figure 4.1 as a steady shift of the minimum of the error rate curves to the left. For example, for 4 nodes, $H$ must be around 20 before the error rate is below 5%, whereas with 75 nodes, $H$ needs to be only 0.5 to reach a comparable level of performance. This behavior is expected since a given space can either be covered by many narrow basis functions, or a few wide basis functions. In addition, the range of widths over which the performance is good increases as the number of nodes increases, as evidenced by the increasingly flatter plateaus of the error curves.

### 4.1.2 Optimal Number of Basis Functions

The complexity of the RBF classifier can be measured by the number of nodes in the entire network structure. Since the number of input nodes and the number of output nodes are defined by the particular problem, there is only one variable that can affect the complexity of the classifier, namely the number of basis function nodes in the middle layer of the network.

For a simple problem like the bullseye problem shown in Figure 4.1, the number of basis functions used does not seem to have an important effect on the performance. Using only 4 basis function nodes, the classifier can achieve performance comparable to the performance obtained using 30 or 60 nodes, provided that the basis functions are wide enough.

For more complex problems, like the vowel problem or the digit problem, the number of basis functions has a much more important impact. Given too few basis functions, the RBF classifier can not form the required decision boundaries for these problems regardless of the width of the basis functions. For complex problems, where fine grain decision boundaries have to be formed, not only must the basis functions be sufficiently narrow, as discussed above, but there also must be enough basis functions.
Figure 4.3: Error Rate of the RBF Classifier on (A) the Test Set and (B) the Training Set of the Digit Problem as a Function of $H$ and the Number of Basis Function Nodes.
This behavior can be seen in figures 4.2 and 4.3. In Figure 4.2, with 4 basis function nodes, the error rate on the vowel problem can only reach 55%. With 8 nodes, the error rate is able to drop to 30%. It is not until there are at least 15 to 30 nodes, that the error rate of the classifier can reach 20%. For the digit problem, shown in Figure 4.3, performance continues to improve until there are at least 30 basis function nodes.

In general, the performance of the RBF classifier improves as the number of basis function nodes increases. There is a certain point, however, after which increasing the number of nodes does not result in a significant improvement of performance. For the vowel and digit problems, the performance levels off after 30 nodes. For the vowel problem, there is surprisingly very little difference between the performance obtained using 75 nodes compared with using 30 nodes. After a sufficient number of nodes is reached, more nodes contribute little.

The experiments performed with the RBF classifier in this study show that there is very little divergence in performance between the training set and the testing set. Parameters that minimize the error in the training set also seem to perform well on the test set. Figure 4.3 shows the performance of the RBF on the digit problem for both the training set and the testing set. As the number of basis function nodes increases, the performance improves on both the training and testing sets. As the widths of the basis functions are varied by changing the global proportionality factor $H$, the performance curves for both the training and testing sets show the same behavior. There is no instance where improving the performance on the training set results in lowering the performance on the testing set. This behavior may be due to the type of decision regions formed by the RBF classifier. The boundaries are formed by an interpolation of relatively smooth Gaussian shaped functions. It is very difficult for this type of framework to form highly specific boundaries that capture only a few spurious data points in the training set. The decision regions tend to be very smooth and as a result generalize well to unseen data.

Because of this behavior, the number of basis function nodes and the appropriate values of $H$ for the RBF classifiers in this study were chosen by simply examining the performance of the RBF classifier on the training set and choosing the number
4.1. RADIAL BASIS FUNCTION CLASSIFIER

<table>
<thead>
<tr>
<th>Problem</th>
<th>Bullseye</th>
<th>Disjoint</th>
<th>Vowel</th>
<th>Digit</th>
</tr>
</thead>
<tbody>
<tr>
<td>$H$</td>
<td>20.0</td>
<td>3.0</td>
<td>0.8</td>
<td>20.0</td>
</tr>
<tr>
<td># Basis Functions</td>
<td>30</td>
<td>30</td>
<td>30</td>
<td>30</td>
</tr>
</tbody>
</table>

Table 4.1: Number of Basis Functions and Values of $H$ Chosen for the RBF Classifiers for the Four Problems.

of nodes and a corresponding value of $H$ that fell in the middle of the range of good performance. Table 4.1 lists the number of basis functions and the corresponding values of $H$ chosen for the RBF classifier for the four problems.

4.1.3 Determining the Basis Function Centers

In the experiments described above, the centers or means, $\mu_i$, of the basis functions were determined by performing K-means clustering on the entire training set and then using the K-means cluster centers as basis functions centers. There are, however, many different ways to determine the basis function centers. In this study, three different methods of determining the set of mean vectors for the basis functions were investigated. They include

(A) Using the K-means algorithm to cluster the training set and using the cluster centers as basis function centers,

(B) Using the Estimate-Maximize (E-M) algorithm to cluster the training set and using the cluster centers as basis function centers, and

(C) Randomly choosing vectors from the training set and using them as basis function centers.

Figure 4.4 plots the error rate of the RBF classifier, using 30 basis function nodes, on the vowel problem as a function of $H$ for the three different methods of determining the location of the basis function centers. The error rate curve for the K-means clustering is marked with triangles ($\triangle$) while the curve for the E-M algorithm is
Figure 4.4: Error Rate of the RBF Classifier on the Vowel Problem For Different Methods of Determining Basis Function Centers.

marked with diamonds (◇). The error rate curve for the randomly chosen basis functions centers is the average of 10 independent runs and is marked with squares (□).

For large basis function variances, there is no significant difference between the three curves. As the variances get smaller (ie. as the basis functions get narrower), however, the method of determining the locations of the basis function centers begins to have a greater effect. For $H$ values above 0.5, all three curves are essentially the same. For $H$ values below 0.5, the performance obtained using randomly chosen basis function centers is much worse than that obtained using a clustering technique. This behavior can be explained by looking at the locations obtained using the different techniques. Using the clustering techniques, the center sample the input data distribution evenly. Choosing vectors randomly from the training set and using them as basis function centers, however, does not guarantee even sampling. All the centers may be concentrated in one area or spread very far apart. With very wide basis
4.1. **RADIAL BASIS FUNCTION CLASSIFIER**

Figure 4.5: Error Rate of the RBF Classifier on the Digit Problem For Different Methods of Determining Basis Function Centers.
functions, the exact location of the basis function centers does not matter much because there is sufficient overlap to form the required discriminant functions. If the basis functions are very narrow, however, the locations of the basis function have a greater effect. If the basis functions are evenly distributed throughout the space, then they can overlap and cover the space with much smaller variances than if they are all concentrated together or spread out far apart.

Figure 4.5 plots the error rate of the RBF classifier, using 30 basis function nodes, on the digit problem as a function of $H$ for the three different methods of determining the location of the basis function centers. Again, the performance for the three different methods is similar for large $H$ values, but diverges for smaller $H$ values. This time, there is a greater divergence between using the E-M algorithm and using K-means clustering. Much better performance is obtained using the E-M algorithm when the basis functions are narrow, indicating that for high dimensionality problems, the E-M algorithm may provide more evenly distributed centers than K-means clustering. This improvement, however, occurs only in regions where the error rates are much higher than the minimum error rate that can be achieved with the RBF classifier. In the regions of interest, where the error rate is very low, there is no improvement. For all practical purposes, there does not seem to be any significant difference between using K-means clustering or the E-M algorithm. For the remaining simulations using the RBF classifier, K-means clustering of the entire unlabeled training set was used to determine the basis function centers.

### 4.1.4 Multi-Scale Basis Functions

As discussed above, the performance of the RBF classifier depends heavily on the proper width of the basis functions. Basis functions that are too narrow or too wide will result in poor performance. With the above framework, where there is a global proportionality factor $H$ that can be used to rescale the basis function widths, an optimal range of $H$ values must be found using computationally expensive methods like cross validation.

One obvious way to eliminate this search is to place multiple basis functions with widths that span a wide range of values on each center and let the weights
determine the optimal combination to use. Although this method may eliminate the need to search for optimal widths, using multiple basis functions on each center essentially translates to increasing the total number of basis function nodes and weight parameters and, as a result, requires more computation. Having too many basis function nodes and a limited amount of training data can lead to poor performance because there may not be enough data to reliably estimate the increased number of weights.

A simple method can be used to reduce the number of basis functions by pruning away basis function nodes that are not useful in forming the decision regions. Recall from Section 2.1, that a correlation matrix, \( R \), of the basis function outputs across the entire training set has to be computed in order to determine the optimal set of output weights, \( w^* \). This correlation matrix can be used to determine, to a first order, which basis functions are important in forming the final decision boundaries, and which are not. The diagonal entries of the correlation matrix \( R \) correspond to the autocorrelation of the basis function outputs across the training set. A high value along the diagonal implies that many training patterns fall close to that particular basis function, while a low value means that very few patterns are close. Usually, basis functions located far away from the data, or basis functions that are very narrow have small autocorrelation values. These basis functions are less likely to be useful in determining the decision boundaries. Thus, a simple heuristic is to eliminate those basis functions that have an autocorrelation value less than some threshold \( \tau_{\min} \).

In addition to the diagonal entries of the correlation matrix, entries off the diagonal can also provide information that can be used to eliminate basis function nodes. Specifically, entry \( R_{ij} \) corresponds to the cross-correlation of the outputs of basis function nodes \( i \) and \( j \) across the training set. A high value for entry \( R_{ij} \) implies that the basis functions \( i \) and \( j \) are highly correlated, while a small value implies that they are not correlated. High correlation between two different basis functions results when the basis functions "cover" the same set of training patterns, which can be interpreted as having the two basis functions performing redundant tasks. One of the two highly correlated basis functions can be eliminated. In this work, the narrower basis function was eliminated. Thus, another simple heuristic is to find all pairs of
basis functions that have a cross-correlation value greater than some threshold $\tau_{\text{max}}$, and eliminate the narrower basis function.

Figures 4.6 and 4.7 show the error rates of the normal and the multiscale RBF classifiers on the disjoint and vowel problems, respectively, as a function of the number of basis function centers. There are three curves in each plot. The curve marked with the squares (□) corresponds to the performance attained by searching for the optimal value of $H$. The curve marked with the triangles (△) corresponds to the performance of the multiple scale RBF classifier using four different width basis functions per center. These four basis functions have widths that are 1, 5, 10, and 50 times that of the initial estimate of $\sigma^2$. If the number of basis function centers is 30, then using 4 multiscale basis functions per center translates to having a total of 120 basis function nodes. Finally, the curve marked with the plus sign (+) corresponds to the performance of the multiple scale RBF classifier using the two heuristic pruning rules described above to eliminate some of the basis function nodes. The thresholds $\tau_{\text{min}}$
Figure 4.7: Error Rate of the Multiscale and Normal RBF Classifiers on the Vowel Problem as a Function of the Number of Basis Function Centers.
and $\tau_{max}$ were set arbitrarily at 0.033 and 0.66 on a normalize scale where the entries in the correlation matrix $R$ can range from 0.0 to 1.0. Across the different problems, these threshold settings eliminated 25% to 60% of the basis functions used in the multi-scale RBF classifier with no pruning.

For these two problems, the performance of the multi-scale RBF, the pruned multi-scale RBF, and the conventional RBF classifiers does not differ significantly. This suggests that given a set of basis functions with varying widths, the weights can be trained to select the optimal combination of basis functions. In addition, the two simple heuristic pruning rules seem to work in the sense that a large number of basis functions that are not useful in performing the classification are eliminated.

There may be a discrepancy in the performance of the multi-scale and conventional RBF classifiers for small numbers of basis function centers. In these cases, the performance of the multi-scale RBF classifier may be better than that of the conventional RBF classifier because the multi-scale RBF classifier has more nodes than the conventional RBF classifier. In addition, for small enough numbers of centers, the conventional RBF classifier may not yet have enough nodes to form the required decision regions.

Figure 4.8 shows the error rate of the multi-scale, pruned multi-scale, and conventional RBF classifiers on the training and testing sets of the digit problem. In this case, there is a large difference in performance between the multi-scale and conventional RBF classifiers. Both the multiscale and pruned multi-scale classifiers do not perform as well as the conventional RBF classifier on the test set. On the training set, however, all three perform equally well. The difference in performance on the test set may be caused by the increased number of weights that have to be estimated for the multi-scale RBF classifiers and the relatively small size of the training set. In other words, there are too many weights and not enough training data to estimate them reliably. With the heuristic pruning, some of the basis function nodes are eliminated, thereby reducing the number of weights and improving the performance, but there are still too many nodes left, and as a result, the performance is still worse than that of the conventional RBF classifier. Although it was not investigated, additional pruning of the basis function nodes may improve the performance by further reducing the
Figure 4.8: Error Rate of the Multiscale and Normal RBF Classifiers on (A) the Test Set and (B) the Training Set of the Digit Problem as a Function of the Number of Basis Function Centers.
number of weights that have to be estimated. However, the additional pruning may also leave the classifier with a too restricted set of basis functions that may not provide enough flexibility for the classifier to be able to perform well.

4.2 High Order Polynomial Network Classifier

4.2.1 Determining Appropriate Polynomial Functions

As described in Section 2.2, the high order polynomial network is made up of simple function nodes that form polynomials from combinations of input variables taken from the training data. The types of polynomials that can be used can vary widely and usually depend on the particular problem. In this study, a variety of different polynomial functions were explored, but as discussed in Section 2.2, not all of them worked well. Preliminary experiments indicate that only two polynomials:

\[ P(v_1) = w_0 + w_1v_1 + w_2v_1^2 + w_3v_1^3 \]  \hspace{1cm} (4.1)

and

\[ P(v_1, v_2) = w_0 + w_1v_1 + w_2v_2 + w_3v_1v_2 + w_4v_1^2 + w_5v_2^2 + w_6v_1^3 + w_7v_2^3 \]  \hspace{1cm} (4.2)

were frequently used. As a result, these were the only ones used in the GMDH networks in this study.

4.2.2 PSE Error Function

In Section 2.2.3, two different error functions which can be used to determine the complexity of the GMDH network were described. One was the regularity criterion, \( r_k^2 \), and the other was the predicted squared error, or PSE, criterion. This Section describes experiments using the PSE criterion to construct a GMDH network. The next section focuses on the \( r_k^2 \) criterion.
Penalty Term Error Variance

Recall that the PSE error function, defined in equation 2.24 and repeated here,

\[ PSE_k = \frac{1}{N} \sum_{p=1}^{N} [d(p) - z_k(p)]^2 + \frac{2\sigma_p^2 W}{N} \]  (4.3)

is made up of the sum of two distinct terms. The first is the mean squared error on the training set between the outputs of the polynomial, \( z_k(p) \), and the desired outputs, \( d(p) \), of 0’s and 1’s, and the second is a penalty term, that penalizes overly complex models, where complexity is measured by the number of coefficients, \( W \), in the final output polynomial. Recall also that there is an error variance, \( \sigma_p^2 \), in the penalty term which can be adjusted to allow for models of various complexity. A large variance will force a simpler model, while a small variance will allow a more complex model. In addition, having a large number, \( N \), of training patterns will also allow for a more complex model while having little training data will require a simpler model.

Figure 4.9 is a plot of the error rate of the GMDH classifier on the disjoint problem...
as a function of the complexity of the model, measured in total number of coefficients in the final output polynomial. Each of the data points on the plot corresponds to a particular value of the error variance, $\sigma^2_\hat{y}$. As the model grows more complex (i.e. $\sigma^2_\hat{y}$ gets smaller), the performance on the training set and on the testing set both improve. This improvement is initially very rapid, but levels off once a certain complexity is reached and then stays relatively constant. From Figure 4.9, the error rate on the test set drops from 9% down to 3% when the number of coefficients is increased from 20 to 100. After 100 coefficients, however, the error rate stays relatively constant at around 3%. Increasing the complexity of the classifier four fold, going up from 100 to 400 coefficients, only reduces the error rate by 1%.

In the case of the disjoint problem, where the decision boundaries are relatively simple, and the training set is a good representation of the test set, there is no divergence in performance between the training and test sets as the complexity of the model is increased. However, on more difficult problems, like the vowel problem, where the
4.2. HIGH ORDER POLYNOMIAL NETWORK CLASSIFIER

decision regions are more complex and the different classes overlap, increasing the complexity of the model does result in a divergence in the performance between the training and test sets. This behavior can be seen in Figure 4.10, which plots the error rate of the GMDH classifier on the vowel problem as a function of the complexity of the model. Initially, as the model complexity increases, the performance on the training set and on the test set both improve. As the complexity is increased further, the performance on the training set continues to improve. The performance on the test set, however, reaches a minimum, and then degrades.

When there is a divergence in performance between training and test sets, it is very important to be able to determine the “proper” complexity of the classifier so that it can provide good performance, but at the same time avoid overfitting the training data. When the PSE error criterion is used to automatically determine the complexity of the GMDH classifier, specifying the maximum size or complexity of the classifier involves determining an appropriate value for the error variance $\sigma_p^2$ term in the PSE error criterion.

In Section 2.2, an estimate of $\sigma_p^2$ using the 0 and 1 class labels was shown to give a poor estimate of the variation of the data. A better estimate of the variation of the data can be obtained by performing a $K = 1$ nearest neighbor (KNN) classification on the training set, and using that error rate as an estimate of $\sigma_p^2$. To do this, the training set is divided into two sets, an exemplar set and an evaluation set. The exemplar set is stored in the KNN classifier as the exemplar database and the evaluation set is classified to obtain the classification error rate. The final $\sigma_p^2$ is the average error rate of several independent classification runs with different evaluation and exemplar sets. Using the performance of a KNN classification of the training set provides a variance measure that is proportional to the complexity of the decision regions of the training set. Problems with complex decision regions will yield higher estimates of $\sigma_p^2$, while problems with simple decision regions will yield lower estimates of $\sigma_p^2$.

Table 4.2 lists the values of $\sigma_p^2$ determined using the KNN method and the resulting number of coefficients in the model for the bullseye, disjoint, vowel, and digit problems.

As expected, the estimates of $\sigma_p^2$ for the more complex problems are larger than
Table 4.2: Estimates of $\sigma_P^2$ Determined Using a KNN Classification on the Training Set and the Resulting Number of Coefficients in the Final GMDH Output Polynomials.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Bullseye</th>
<th>Disjoint</th>
<th>Vowel</th>
<th>Digit</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\sigma_P^2$</td>
<td>0.012</td>
<td>0.032</td>
<td>0.05</td>
<td>0.046</td>
</tr>
<tr>
<td># Coefficients</td>
<td>76</td>
<td>100</td>
<td>328</td>
<td>150</td>
</tr>
</tbody>
</table>

those for the simpler problems. It may seem counter-intuitive to have high $\sigma_P^2$ variances, hence *simpler* models, for the complex problems and low $\sigma_P^2$ variances, hence *more complex* models for the simpler problems. Simpler models are required for complex problems to ensure that the models do not overfit the training data. Overfitting in a complex problem may result in *worse* performance than overfitting in a simple problem because the training set is less likely to be as good a representative of the test set. Given enough data, however, complex models can be formed because the $\frac{W}{N}$ factor in the penalty term, will be able to balance out the effect of a large $\sigma_P^2$ variance.

For the bullseye, disjoint, and vowel problems, the KNN method of selecting $\sigma_P^2$ gives a reasonable estimate of the error variance. The only exception is with the digit problem, where the estimated $\sigma_P^2$ variance is *too small*. The resulting classifier overfits the training data and, as a result, performs poorly on the test set. This can be attributed to the small amount of training data available for the problem. A simpler model, one with a higher $\sigma_P^2$, is needed in order to avoid overfitting. The appropriate error variance for the digit problem was found to be $\sigma_P^2 = 0.12$, which allowed models with an average of 78 coefficients.

**Number of Layers**

Once the value of $\sigma_P^2$ is determined, the PSE error criterion can be used to evaluate, rank, and select the polynomials at each layer to be passed on as inputs to the next layer, as described in Section 2.2. In addition, the PSE error criterion is also used
Figure 4.11: (A) The Predicted Squared Error and (B) Error Rate of the GMDH Classifier on the Vowel Problem as a Function of the Number of Layers.
to determine when to stop the growth of the classifier network. A network stops growing when the lowest PSE measure of the current layer is greater than the lowest PSE measure of the previous layer. Thus, once the error variance $\sigma^2$ is set, the GMDH algorithm is essentially self-organizing, in the sense that it will automatically determine which combinations of polynomials to use and how complex to grow.

Figure 4.11 shows a plot of the PSE error and the error rate of the GMDH classifier on the vowel problem as a function of the number of layers in the network. The PSE values plotted are the lowest PSE measures of each layer. The value of the PSE error continues to decrease at every new layer until it either remains the same or increases slightly, at which point, the network stops growing. For the vowel problem, the PSE error at layer 9, is slightly larger than the PSE error at layer 8, causing the network to stop growing after 8 layers. In addition to the decreasing PSE error criterion, the error rates on the training and test sets also decrease as the number of layers increase. Most of the improvement occurs in the first few layers of the network. In the vowel problem, the error rate drops from 35% down to roughly 22% going from layer 1 up to layer 4. After layer 4, the error rate levels off and stays relatively constant at around 22%. Going all the way to layer 8, the error rate dips only slightly to 21.9%.

In general, the PSE error criterion seems to be able to control the growth of the network so that the model does not overfit the training data. This ability is shown by the fact that both the training and test set performances improve with each new layer, and that the two performances follow each other closely. There is no instance of divergence in performance where the error rate on the training set drops rapidly to zero while the error rate on the test set stays relatively high.

**Width of Each Layer**

For the GMDH classifier, the number of variables that are passed on to the next layer or the width of each layer is a free parameter that can be adjusted. There are many different methods for determining the proper width of each layer [2]. One method is to fix the maximum width of all the layers so that only the top $Q$ variables or polynomials
are kept at each layer. Another is to slowly reduce the number of variables allowed as the number of layers increases. The rationale behind this approach is to slowly narrow down the set of possible polynomials to a small set that have good performance. Yet another method is to not put a specific limit on the number of variables at each layer, but to keep only those polynomials that have a PSE error less than a certain threshold. This threshold condition essentially constrains the number of variables kept. For simplicity, the first method of setting a maximum width for all the layers of the network was used for all the GMDH classifier simulations.

In general, increasing the maximum width of each layer improves the performance of the classifier. This is true, however, only up to a certain point. Shown in Figure 4.12 is a plot of the error rate of the GMDH classifier on the disjoint problem as a function of the maximum number of variables in each layer. Increasing the width of each layer from 5 to 18 variables lowers the error rate from 5.5% down to 2.8%. After a width of 18, however, further increases in the width do not change the performance of the
Figure 4.13: Error Rate of the GMDH Classifier on the Vowel Problem as a Function of the Width of Each Layer.

classifier; the error rate remains at 2.8%. The vowel problem, shown in Figure 4.13, shows a similar behavior. The performance of the classifier increases until a width of 16 is reached, after which the error rate levels off at around 22%.

Improvement in performance as the width of each layer is increased is expected because increasing the number of variables at each layer increases the total number of possible variable combinations available which, in turn, increases the potential of forming better polynomial functions. At a certain point, however, after all the "useful" polynomials have already been included in the layer, adding additional polynomials does not improve the performance of the classifier. As mentioned above, for the disjoint problem the limit seems to be at 18 variables per layer, while for the vowel problem the limit seems to be at 16 variables per layer.

The more polynomials there are in each layer, the more time the training process will require. Remember that new polynomials at each layer are formed from all possible combinations of the variables of the previous layer taken one or two at a time. This means that a doubling of the number of variables in each layer will increase the
number of new polynomials that have to be evaluated at each layer almost four fold! In general, the number of polynomials that have to be constructed and evaluated is order $O(Q^2)$, where $Q$ is the width or the number of variables in each layer. Thus, it is not wise to have overly large widths. For the problems investigated in this study, the width at which the performance ceases to increase was the one used to construct the classifiers.

4.2.3 Regularity Criterion Error Function

The regularity criterion, defined in equation 2.23 and repeated here,

$$r_k^2 = \frac{\sum_{p \in S_{tr}} (d(p) - z_k(p))^2}{\sum_{p \in S_{tr}} d^2(p)}, \quad k = 1, \ldots, \frac{1}{2}Q(Q - 1) + Q$$

(4.4)

is another error function, like the PSE error criterion, that can be used to construct a GMDH network classifier. As mentioned in Section 2.2, computing the regularity criterion, requires splitting the training set into two separate parts: an evaluation subset and an estimation subset. The regularity criterion, is then computed from a simple cross validation training and evaluation process, described in Section 2.2, using these two subsets. Any time a training set is required to be split up, there will be differences in performance that are dependent on the particular composition of the subsets. Thus, in order to obtain a reliable estimate of the performance, many different subset combinations have to be evaluated, meaning that a cross validation (CV) technique must be used.

For the bullseye, disjoint, and vowel problems, a four-fold cross validation method was used to estimate the performance of the GMDH classifier. For the digit problem, however, only a two-fold cross validation was used because of the small amount of training data. To perform four-fold cross validation, the training set, $S$, is initially divided into four mutually exclusive subsets, $S_A, S_B, S_C,$ and $S_D$. These four subsets are “balanced” in the sense that each subset contains approximately the same number of data vectors and each has vectors from the different classes in the same proportion. Four different classifiers are then constructed, each one using three out of the four subsets as the estimation subset, $S_{es}$, to estimate the polynomial coefficients, and the
one remaining subset as the evaluation subset, $S_{ev}$, to compute $r_k^2$. For example, one classifier will use subsets $S_A$, $S_B$, and $S_C$ to estimate the polynomial coefficients, and subset $S_D$ to evaluate $r_k^2$. Another will use subsets $S_A$, $S_B$, and $S_D$ to estimate the polynomial coefficients, and subset $S_C$ to evaluate $r_k^2$. The other two classifiers will use the other two possible combinations of the subsets in order to build the network. After these four classifiers are built, they are then used to classify the test data. The estimate of the performance of the GMDH classifier is then determined by taking the average performance of the four classifiers.

Number of Layers

As with the PSE error criterion, the regularity error criterion is used at every layer to evaluate, rank, and select the polynomials to be passed on as inputs to the next layer. It is also used to determine when to stop the growth of the network. A network stops growing when the lowest regularity error measure (rmin) of the current layer is greater than the rmin of the previous layer. Thus, the GMDH algorithm using the regularity criterion should be self-organizing, since provisions are provided for selecting which polynomials to use in each layer and for determining when to stop the network growth. However, on the problems investigated in this study, the regularity condition was not a strong enough criterion to stop the growth of the network at a reasonable level. Usually, it allowed overly complex models with more than 20 layers to be constructed.

One possible solution to this problem is to use a two stage process to build the GMDH classifier. The first stage is to determine the proper size or complexity for the network, and the second stage is to use this new information to construct the classifier, stopping the growth of the network once it has reached the proper size. The 1 standard error or 1 SE rule is a heuristic described on page 80 in CART [5] that can be used in the first stage of the process to determine an appropriate size for the network. The first step of the 1 SE rule is to perform cross-validation on the training set and to compute the average training set error rate from the set of classifiers constructed. The next step is to find the lowest error rate, $E_{min}$, on the average error rate curve and note the corresponding layer number, $L_{min}$. A binomial
Figure 4.14: (A) The Regularity Error (rmin) and (B) the Error Rate of the CV GMDH Classifier on the Vowel Problem as a Function of the Number of Layers.
Table 4.3: Determining The Proper Complexity for the CV GMDH Classifier.

\[
\begin{array}{|c|c|c|c|c|c|c|}
\hline
\text{Class} & \mathcal{E}_{\text{min}} & L_{\text{min}} & \sigma_{SE} & \mathcal{E}_{\text{opt}} & L_{\text{opt}} & N_{\text{train}} \\
\hline
\text{Bullseye} & 1.00\% & 10 & 0.44\% & 1.44\% & 8 & 500 \\
\text{Disjoint} & 3.65\% & 10 & 0.83\% & 4.48\% & 7 & 500 \\
\text{Vowel} & 25.88\% & 9 & 2.38\% & 28.27\% & 7 & 338 \\
\text{Digit} & 0.93\% & 10 & 1.15\% & 2.08\% & 5 & 70 \\
\hline
\end{array}
\]

\textit{standard deviation}

\[
\sigma_{SE} = \sqrt{\frac{\mathcal{E}_{\text{min}}(1 - \mathcal{E}_{\text{min}})}{N_{\text{train}}}}
\]  

(4.5)

is then computed from the training set data. In equation 4.5, \( N_{\text{train}} \) is the total number of vectors in the training set. The appropriate size of the network is then determined by setting the maximum layer to be \( L_{\text{opt}} \), where \( L_{\text{opt}} \) is the \textit{minimum} layer, with error \( \mathcal{E}_{\text{opt}} \), satisfying

\[
\mathcal{E}_{\text{opt}} \leq \mathcal{E}_{\text{min}} + \sigma_{SE}.
\]  

(4.6)

In other words, the appropriate layer is the \textit{minimum} layer that has an error rate \textit{just} lower than one standard deviation above that of the lowest error rate.

For example, Figure 4.14 shows the error rate of the GMDH classifier using the regularity error criterion on the vowel problem as a function of the number of layers. The curve plotted for the training set error (\( \Delta \)) is the average error of the 4 classifiers constructed using the four-fold cross validation method described above. The lowest average error, for this problem, is 25.8% which occurs at layer 9. The standard deviation, \( \sigma_{SE} \), from equation 4.5, with a training set size of \( N_{\text{train}} = 338 \), is 2.38%. The layer \( L_{\text{opt}} \), at which equation 4.6 is satisfied, is layer 7. Thus, the GMDH network for the vowel problem should only be allowed to grow to a maximum of 7 layers. Table 4.3 lists the error rates and layer numbers derived from the 1 SE rule and cross-validation experiments with the four data sets to determine the proper complexity of the GMDH classifier.

Once the proper complexity of the classifier is determined, the cross-validation classifiers can then be reconstructed, this time constraining the \textit{maximum} number:
4.2. **HIGH ORDER POLYNOMIAL NETWORK CLASSIFIER**

<table>
<thead>
<tr>
<th></th>
<th>4x CV</th>
<th>All Data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Train</td>
<td>Test</td>
</tr>
<tr>
<td>Bullseye</td>
<td>1.4%</td>
<td>1.1%</td>
</tr>
<tr>
<td>Disjoint</td>
<td>3.7%</td>
<td>4.7%</td>
</tr>
<tr>
<td>Vowel</td>
<td>27.8%</td>
<td>27.7%</td>
</tr>
<tr>
<td>Digit</td>
<td>1.9%</td>
<td>25.5%</td>
</tr>
</tbody>
</table>

Table 4.4: Error Rates for the CV GMDH and “All Data” GMDH Classifiers.

of layers in the network to be \( \mathcal{L}_{opt} \). These classifiers can then be used to classify the data in the test set and their performance averaged to obtain an estimate of the classification accuracy of the classifier. In addition to reconstructing the cross-validation classifiers, a new classifier that uses all the training data in both the estimation and the evaluation subsets can be constructed. The hope here is that using all the data will allow the weight parameters to be estimated more robustly and will lead to a better classifier. It turns out, however, that the classifiers trained using all the training data in both the evaluation and estimation subsets perform well on the training set, but do not perform well on the test set. Table 4.4 shows the training and test set error rates on the four problems when GMDH classifiers constructed using the four-fold CV method discussed above are used, and when classifiers constructed using all the training data in both the evaluation and estimation subsets are used.

For all four problems, the error rate on the training set is lower when the GMDH classifier was trained using all the training data in both the evaluation and estimation subsets than when four-fold cross validation was performed. The error rates on the test set, however, do not show a corresponding decrease. In fact, for three out of the four problems, the error rate on the test set shows an increase. This divergence in performance occurs because the entire training set is used to both estimate and select the set of polynomials. Given this situation, the training algorithm will probably construct a classifier that overfits the training set. This classifier will, of course, be able to perform well on the training set, but it may or may not be able to perform well on unseen test set. If the training set is a good representative of the test set,
Figure 4.15: Error Rate of the CV GMDH Classifier on the Vowel Problem as a Function of the Width of Each Layer.

as in the bullseye and disjoint problems, then the divergence is not significant. If, however, the training set is not representative of the test set, as in the vowel and digit problems, then the divergence is more evident. Even with the limited number of layers, upper bounded by $L_{opt}$, the classifier is still able to overfit the training data. Given more layers, the classifier will overfit the training data even more. Because of this divergent behavior when trying to use all the training data in both the estimation and evaluation subsets, the four-fold cross validation method was used for all GMDH classifiers using the regularity error criterion.

**Width of Each Layer**

Figure 4.15 plots the error rates of the GMDH classifier trained using the regularity criterion on the vowel problem as a function of the maximum width of each network layer. Unlike the case with the PSE error criterion, the GMDH classifier trained using the regularity criterion shows a divergence in the performance between the training
set and the testing set as the width of the network layers is increased. Initially, both the test and training set error rates decrease, but after a certain point, the two error rates diverge. This indicates that the regularity error condition is not able to select the right polynomials needed to build a classifier that generalizes well. Given more polynomial combinations to work with, the regularity error criterion selects those that minimize the training set error rate, leading to a classifier that overfits the training data. For the problems investigated in this study the width at which the training and test set error rates just begin to diverge was used to construct classifiers.

### 4.2.4 Comparison of the PSE and Regularity Error Functions

Although the building blocks (i.e., set of polynomials) for the GMDH network classifier are the same whether the PSE error criterion or the regularity error criterion is used, the final classifier constructed and the behavior during network growth using these two different error criteria are very different.

Figure 4.16 shows the error rates of a GMDH classifier trained using the regularity error criterion (Δ) and the PSE error criterion (□) on the digit problem. Increasing the width of the network layers causes the test set error rate of the GMDH classifier using the regularity criterion to increase. For the GMDH classifier using the PSE criterion, however, the test set error rate stays relatively constant regardless of the width of each layer. Since the types of functions available to both classifiers are identical, this difference in performance can be attributed entirely to the ability of the error function to select the proper polynomial variables. The PSE criterion is clearly better than the regularity criterion for selecting the right set of polynomials to use at each layer. This is also reflected in the performance of the classifier as the network grows layer by layer. Figure 4.17 plots the error rates of GMDH classifiers designed using the PSE error criterion (□) and the GMDH classifier using regularity error criterion (Δ) for the vowel problem, as a function of the number of layers in the network. The PSE error criterion is consistently able to perform better than the regularity error criterion at every layer of the network. This indicates, again, that
Figure 4.16: Error Rates of GMDH Classifiers Designed Using the CV and the PSE Error Criterion on the Digit Problem as a Function of the Number of Variables at Each Layer.

Figure 4.17: Error Rates of GMDH Classifiers Designed Using the CV and the PSE Error Criterion on the Vowel Problem as a Function of the Number of Layers.
the PSE error criterion is able to form a better set of polynomials than the regularity criterion.

### 4.3 Linear Decision Tree Classifier

The linear tree classifier algorithm described in Section 2.3 is made up of two distinct stages. The first stage builds an overly complex tree classifier that is designed to perform *perfectly* on the training set. As shown in Figure 4.18, which plots the error rate of the linear tree classifier on the disjoint problem as a function of the number of nodes in the tree, the error rate on the training set goes to zero as the number of nodes in the classifier increases. Since this classifier *overfits* the training data, it may or may not be able to perform well on the unseen data in the test set. Once again, this ability depends on how well the training set represents the test set. For the
simple disjoint problem in Figure 4.18, the test set error decreases with the training set error as the number of nodes in the tree increases. However, for the more complex vowel problem, which is shown in Figure 4.19, the test set error rate diverges from the training set error rate as the number of nodes gets very large. Initially, there is improvement in the performance of the test set as the number of nodes increases, but after a certain point, in this case 30 nodes, the improvement levels off, and then begins to get worse as the number of nodes gets large.

The second stage of the linear tree classifier algorithm successively prunes nodes from the overly complex tree formed in the first stage in order to form a smaller size tree. As discussed in Section 2.3, a strength measurement, \( g(t) \), is computed for each non-terminal node \( t \). This measure is then used to eliminate the nodes that are not important or efficient in forming the classification boundaries. As more nodes are pruned from the tree, the performance on the training set, of course, gets worse. The performance on the test set, however, may stay relatively constant, or improve, as in
the case of the vowel problem, or it may follow the training set performance and get worse, as in the case of the disjoint problem. One of the main concerns in pruning away nodes is determining the appropriate number of nodes at which to stop the pruning process so as to obtain a small tree that can achieve good performance on both the training and test sets.

4.3.1 Number of Nodes

For simple problems, like the bullseye and disjoint problems, where the training set is a good representation of the test set, and the training and test set error rates follow each other closely, the appropriate number of nodes to prune to can be determined by eliminating nodes until the test set error rate starts to increase. For the disjoint problem shown in Figure 4.18, the appropriate number of nodes to prune to is 29. For the bullseye problem, the proper number of nodes is 17. For cases where the training and test set error rates diverge, however, a different method for determining the proper number of nodes to prune to must be used. The heuristic 1 SE rule described in CART [5], which was used previously to determine the proper number of layers in the GMDH network classifier, can also be used to determine the proper number of nodes in the linear tree classifier.

For the vowel problem, a ten-fold cross validation was performed and used in the 1 SE rule. The average training set error rate from the ten-fold cross validation is shown in Figure 4.20 as the dashed curve. The minimum average training set error rate, $E_{\text{min}}$, is 26.6% and occurs at node $N_{\text{min}} = 87$. Using $E_{\text{min}}$ and the size of the training set which, for the vowel problem, is equal to 338, the binomial standard deviation, $\sigma_{SE}$, defined in equation 4.5, is then computed. It is equal to 2.55%. Next, the appropriate number of nodes can be determined by finding the minimum number of nodes, $N_{\text{opt}}$, which has an error rate, $E_{\text{opt}}$, that satisfies equation 4.6. In this case, $N_{\text{opt}} = 27$ has an error rate $E_{\text{opt}} \leq 29.16\% = E_{\text{min}} + \sigma_{SE}$.

After this stopping point of 27 nodes has been determined, the entire training set is then used to construct a new over specified tree classifier. This classifier is then pruned back until there are only 27 nodes in the tree. This stopping point is marked in Figure 4.20 by the solid vertical line. A plot of the error rate of the classifier on
Figure 4.20: Stopping Point Predicted by the 1 SE Rule for the Vowel Problem.

the actual test set as a function of the number of nodes is shown as a reference. For the vowel problem, the 1 SE rule is able to select the proper number of nodes so as to achieve good performance on both the test set and the training set.

4.4 Gaussian Mixture Classifier

As mentioned in Section 2.4, many different special case models are often used to estimate the covariance matrix in the Gaussian mixture classifier. These models are usually simpler than the original full covariance matrix in the sense that there are fewer parameters to estimate. Although fewer parameters allows for less general distributions, fewer parameters may be desired either for faster computation or when there is very little training data. One approximation to the covariance matrix that is often used is to assume that the components in the input vectors are independent of each other, implying that the covariance matrix is diagonal. With an input dimension
of $d$, a full covariance matrix has $d + \frac{d^2 - d}{2}$ parameters to estimate whereas a diagonal covariance matrix has only $d$ parameters to estimate. For large dimensionalities, this can significantly reduce the number of parameters.

In addition to reducing the number of parameters in the covariance matrix, there are also other ways to reduce the overall number of parameters that have to be estimated. One method is to restrict all the Gaussian components used in the classifier to have the same shape or covariance matrix. In this case, there is only one grand covariance matrix that has to be estimated for all the component Gaussians across all the different classes. Another method is to use only one set of Gaussian components and let all the different class distributions be formed from this one set. In this case, only one set of means and covariances has to be estimated for all the different classes. In this tied mixture case, the only difference in the set of Gaussian components between the different classes is the set of component weights used to form the linear combination of the Gaussians. A further restriction is to limit all the covariance matrices of the components in the tied mixture set to be the same.

In this study, two variations of three different types of Gaussian mixtures, for a total of six different possibilities, were examined. The three different types of mixtures are described below and include normal, grand, and tied mixtures.
normal  • Each class has its own set of Gaussian components.
         • Each component has its own mean vector.
         • Each component has its own covariance matrix.

grand   • Each class has its own set of Gaussian components.
         • Each component has its own mean vector.
         • Each component has the same covariance matrix.
         • There is only one covariance matrix in the entire model.

tied    • Each class has the same set of Gaussian components.
         • Each component in the set has its own mean vector.
         • Each component in the set has the same covariance matrix.
         • There is only one covariance matrix in the entire model.

The six different possibilities are derived from having one set of these three types of mixtures use diagonal covariance matrices and another set of three use full covariance matrices.

4.4.1 Diagonal and Full Covariance Matrices

As mentioned above, a full covariance matrix allows for more general distributions than a diagonal covariance matrix, but also requires more computation and more training data. Whether or not the extra flexibility provided by a full covariance matrix is needed depends on the complexity of the class distributions of the particular problem and on the assumptions made about the set of Gaussian components used in the model.

In the bullseye problem, for example, using full or diagonal covariance matrices does not make much of a difference when the covariance matrices are normal. This can be seen in Figure 4.21, which plots the error rate of the Gaussian mixture classifier on the bullseye problem for the six different types of mixtures described above. Plot (A) shows error rates for those classifiers using diagonal covariance matrices, while plot
Figure 4.21: Error Rate of the Gaussian Mixture Classifier for the Bullseye Problem Using (A) Diagonal Covariance Estimates and (B) Full Covariance Estimates.
CHAPTER 4. EXPERIMENTAL RESULTS

(B) shows error rates for those using full covariance matrices. The performance of the classifier using mixtures with normal covariance matrices is very similar regardless of whether the covariance matrices are full or diagonal. This behavior indicates that mixture components with either normal-diagonal or normal-full covariance matrices are able to model the bullseye distribution accurately.

The performance of the classifier using mixtures with across class grand covariance matrices is also insensitive to whether the covariance matrices are full or diagonal. The performance of the classifier using these grand covariances, however, is worse than the performance using mixtures with normal covariances when there are small numbers of mixtures. The poorer performance is probably due to the fact that in the grand covariance case, all the components in the model are restricted to have the same covariance matrix. This restriction may be too strong in the sense that there is not enough flexibility to model the class distributions until many component mixtures are available.

With tied mixtures, using full covariance matrices provides much better performance than using diagonal covariance matrices. For this case, the added flexibility provided by the extra parameters seems to be very significant. Using a full covariance matrix, only 4 mixture components are needed in the tied mixture set in order to obtain good performance, whereas using a diagonal covariance matrix requires over 60 components before comparable performance is achieved.

In addition to performance differences between diagonal and full covariance matrix estimates, there are also performance differences between the three normal, grand, and tied mixture types. For the bullseye problem, having 1 mixture component per class is enough to achieve good performance using either normal-diagonal or normal-full covariance matrices. This is because the distribution in the bullseye problem can be modeled perfectly with two Gaussians centered at the origin, each with different variances. For the grand covariance case, however, where all the Gaussian components have the same covariance matrix, many more components are needed before the bullseye class distributions can be modeled accurately. For the tied mixture case, there is a similar problem in that the two class distributions share the same components and will have a difficult time modeling the different distributions with
very few components. As noted above, the task for the tied mixture case is made easier by using a full covariance matrix instead of a diagonal covariance matrix.

In the bullseye problem, there are only two input dimensions, which means that the full covariance matrix has only 4 parameters compared to the 2 parameters of the diagonal covariance matrix. In this case, the reduction in the number of parameters going from full to diagonal covariance matrices is not significant. On the digit problem, however, where the input dimension is 22, the full covariance matrix has \( \frac{1}{2} \times 22 \times 22 \) or 242 parameters whereas the diagonal covariance matrix has only 22 parameters. In this case, the reduction in the number of parameters going from full to diagonal covariance matrices is very significant. As a result of this and the fact that the training set for the digit problem is small, the differences between using diagonal and full covariance matrices in the digit problem are much more dramatic.

Figure 4.22 plots the error rate of the Gaussian mixture classifier on the digit problem for the six different types of mixtures described above. Again, plot (A) shows error rates for those classifiers using diagonal covariance matrices, while plot (B) shows error rates for those using full covariance matrices. For the normal and the grand covariance estimate cases, the performance obtained using the diagonal covariance matrix is much better than that obtained using the full covariance matrix. For the tied mixture estimate, using a full covariance matrix results in error rates greater than 50% (off the scale of Plot B in Figure 4.22) regardless of the number of mixtures, whereas using a diagonal covariance matrix, results in low error rates as the number of component mixtures increases. The differences in performance between using full and diagonal covariance estimates, in the case of the digit problem, illustrates that having large numbers of parameters may result in worse performance if there is not enough training data to estimate the parameters accurately.

For the problems investigated in this study, diagonal covariance matrices provided sufficient flexibility to model all the class distributions accurately. Because of this and the fact that they also reduce the number of parameters that have to be estimated, diagonal covariances were used for the remaining Gaussian mixture classifier simulations.
4.4.2 Number of Mixture Densities

In general, increasing the number of component mixtures allows more general distributions to be modeled, meaning that the performance of the classifier should improve. Increasing the number of component mixtures, however, also increases the number of parameters that have to be estimated. Thus, as in the case of having more complex covariance matrices, improvements in performance can only occur if there is enough training data to reliably estimate the increased number of parameters.

For the bullseye problem, the performance of the classifier improves as the number of component mixtures increases, for the grand covariance and tied mixture cases. The reason for this behavior is two fold. One is that as the number of mixture components increases, the different mixture types are able to model the class distributions better. The other is that there is enough data in the training set to estimate the increasing number of parameters. As mentioned before, the normal covariance case is able to model the class distributions accurately with just one component per class. Because of this, the performance stays level regardless of the number of mixture components.

For the digit problem, as the number of component mixtures increases, the performance of the classifier using grand covariances and tied mixtures improves while the performance of the classifier using normal covariances gets worse. The cause of this degradation in performance lies in the fact that in the normal covariance case, each component has its own covariance matrix, whereas in the grand covariance and tied mixture cases, there is only one covariance matrix for all the components. Increasing the number of component mixtures increases the number of parameters that have to be estimated dramatically in the normal covariance case, but does not affect the grand covariance tied mixture cases as severely. Because of the limited amount of training data in the digit problem, the increased number of parameters can not be estimated accurately, resulting in poor performance.

For the problems investigated in this study, very small numbers of component mixtures were needed in order to model the class distributions accurately provided the mixture type (ie. normal, grand, tied) was chosen correctly. For the bullseye and vowel problems, good performance was obtained using only 1 and 2 component mixtures per class, respectively, when normal covariance matrices were used. For the
Figure 4.22: Error Rate of the Gaussian Mixture Classifier for the Digit Problem Using (A) Diagonal Covariance Estimates and (B) Full Covariance Estimates.
disjoint problem, 16 components per class were required, using normal covariance matrices, in order to achieve good performance. The digit problem performed well with only 2 components per class using grand covariance matrices.

4.5 Condensed K Nearest Neighbor Classifier

4.5.1 Order of Presentation

Given the sequential nature of the training algorithm in the condensed K nearest neighbor (CKNN) classifier, the models that are constructed (ie. the set of vectors that are kept) are very sensitive to the order of presentation of the training vectors to the classifier. Changing the order of the vectors results in a different set of vectors being kept and hence different classifiers being constructed.

The error rates of the CKNN classifier on the vowel problem from ten independent runs with random ordering of the data is shown in Figure 4.23. Error rates are plotted for different numbers of nearest neighbors, K. The number of nearest neighbors, K', used in the training to classify the training vectors for this example, is equal to one. It is clear that different orderings of the training vectors from trial to trial can result in very different performances. In some cases the error rate can differ by as much as 20%.

There are many ways to try to get around this problem. One is to train the classifier by cycling through the set of training vectors multiple times until no new vectors are added to the stored database. This guarantees that all the vectors that are important in forming the classification decisions are given the opportunity to be included in the exemplar database. Typically, no more new vectors are stored after 2 or 3 iterations through the training data.

Another method is to run the training algorithm a number of times on different permutations of the training set, and then take the union of the stored exemplars across the different trials and run it through the algorithm one more time to obtain a set of exemplar vectors for the database. The rationale behind this approach is that over the course of many different random runs, the vectors that are important in
Figure 4.23: Error Rate of the Condensed Nearest Neighbor Classifier on the Vowel Problem as a Function of the Number of Nearest Neighbors for 10 Randomly Initialized Runs.
Figure 4.24: Set of Vectors Selected by the Condensed Nearest Neighbor Classifier for the Bullseye Problem.
4.5. **CONDENSED K NEAREST NEIGHBOR CLASSIFIER**  

<table>
<thead>
<tr>
<th>Bullseye</th>
<th>Method A</th>
<th>Method B</th>
<th>Method C</th>
<th>Original</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bullseye</td>
<td>29</td>
<td>35</td>
<td>35</td>
<td>500</td>
</tr>
<tr>
<td>Disjoint</td>
<td>43</td>
<td>57</td>
<td>49</td>
<td>500</td>
</tr>
<tr>
<td>Vowel</td>
<td>126</td>
<td>161</td>
<td>152</td>
<td>338</td>
</tr>
<tr>
<td>Digit</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>70</td>
</tr>
</tbody>
</table>

**Method A** Run the training set through the CKNN training algorithm once.

**Method B** Run the training set through the CKNN training algorithm multiple times, until no more new vectors are added.

**Method C** Take the union of the stored vectors from 10 independent Method A runs, and run those through the CKNN training algorithm to obtain a final set of exemplar vectors.

Table 4.5: Number of Training Vectors Stored by the Condensed K Nearest Neighbor Classifier for Different Methods of Training.

Forming the classification decisions will be singled out. Taking the union of these points should ensure a good exemplar set. Running this set of vectors through the algorithm serves to reduce the size of the set by eliminating vectors that are redundant. Shown in Figure 4.24 are the 35 (out of 500) training vectors selected for the bullseye problem using this method. Note that most of the selected vectors are located very close to the optimal decision boundary (solid curve), which is where fine grain decisions have to be made. It is interesting to note that, in this respect, the CKNN classifier seems to do almost the exact opposite of what other classifiers, like Gaussian, K-means, and learning vector quantizer (LVQ) classifiers, do [12]. These classifiers store only a few vectors near the center of each class distribution. The main difference between the Gaussian, K-means, and LVQ classifiers and the CKNN classifier is that these classifiers try to model the distribution of the data whereas the CKNN classifier tries to model the decision boundaries. The linear tree classifier also tries to model decision boundaries.
In general the number of vectors stored by the condensed $K$ nearest neighbor classifier using any of these three methods is much less than the number of vectors in the original training set. Table 4.5 lists the number of training vectors stored for the three methods described above.

For a given problem, the number of vectors stored does not differ significantly across the three different training methods. Across the different problems, the trend is for problems with more complex decision regions to have more stored exemplars. For the simple bullseye and disjoint problems, less than 10% of the original training vectors have to be stored. For the more complex vowel and digit problems, somewhere between 25% and 50% of the training vectors have to be stored.

Figure 4.25 shows the error rate of the CKNN classifier on the disjoint and vowel problems as a function of the number of nearest neighbors, $K$, for the three methods of training the classifier. The curves for Method A are the average of ten independent training and classification runs. The number of nearest neighbors, $K'$, used in the training, for all three methods was one. Running the training set through the CKNN training algorithm once (Method A) results in worse performance than either Method B or Method C. The reason, as mentioned above, is because of the sensitivity of the classifier to the ordering of the training vectors. Sometimes, the ordering will result in a high performance classifier, and sometimes the ordering will result in a poor performance classifier. This variability is clear from Figure 4.23. The average performance using Method A will, of course, be worse than the best performance that can be achieved using Method A. In general, there is very little difference between the performance of the CKNN classifier trained using Method B and that trained using Method C. In terms of absolute “best” performance, however, there is a slight difference between Method B and Method C. On the problems investigated in this study, classifiers trained using Method C were able to perform slightly better than those trained using Method B. Take for example, the disjoint and vowel problems shown in Figure 4.25. For the disjoint problem and $K = 1$, the best performance is obtained using Method C. For the vowel problem and $K = 8$, the best performance is also obtained using Method C. For this reason, the remaining simulations using the CKNN classifier were all trained using Method C.
Figure 4.25: Error Rate of the Condensed Nearest Neighbor Classifier on (A) the Disjoint Problem and (B) the Vowel Problem as a Function of the Number of Nearest Neighbors for Three Different Training Methods.
4.5.2 Number of Nearest Neighbors

For the condensed nearest neighbor classifier, there are two different nearest neighbor free parameters. One is the standard $K$, the number of nearest neighbors to use in the voting during classification. The other is $K'$, the number of nearest neighbors to use during training.

Since the vectors stored by the CKNN classifier are not the same as those stored by the conventional KNN or the KD tree classifiers, the performance of the CKNN classifier as a function of $K$ will be different from that described in Section 4.6. In particular, since the CKNN classifier is trying to store only those vectors that are important in forming the decision boundaries, which translates to points that are close to the edges of the decision regions, the voting process used in the traditional KNN classifier is not valid. The original KNN voting process estimates the Bayes posteriori probability and requires all training vectors. In the CKNN case, most of the vectors that play an important role in the voting process, i.e., those that lie in the interior of the class distribution, are eliminated. As shown in Figure 4.24, almost none of the points that lie in the interior of the class regions are kept. Only those points that lie along the boundaries are kept. In order for the CKNN classifier to be able to perform well, only the nearest few points should be checked. Using a large $K$ will take points on the other side of the boundary into account and will result in poor performance.

The error rate of the CKNN classifier on the disjoint problem, which is shown in plot (A) of Figure 4.25 behaves as predicted. For $K = 1$, the performance is good, but for $K > 1$, the performance degrades rapidly. The CKNN classifier exhibits this behavior for the bullseye, disjoint, and digit problems. For the vowel problem, however, the best performance is obtained using $K = 8$. This can be seen in plot B of Figure 4.25. This may be due in part to the complex decision boundaries in the problem and in part to the fact that about half of the original training set is stored in the exemplar database. With this much data, choosing $K > 1$ and using the voting procedure is still be meaningful.

The number of nearest neighbors, $K'$, used in the training of the CKNN classifier was set to one for all the CKNN classifier simulations discussed above. Of course,
Figure 4.26: Error Rate of the Condensed Nearest Neighbor Classifier on the Vowel Problem as a Function of $K'$.

$K'$ can be set to any arbitrary value. Figure 4.26 plots the error rate of the CKNN classifier on the vowel problem as a function of $K$, for three different values of $K'$, specifically $K' = 1, 5$, and $8$. As $K'$ increases, the performance on the training set improves. For $K' = 1$, the training set error rate is zero for $K = 1$, but jumps up to around 30% for the remaining values of $K$. For $K' = 5$, the training set error rate stays relatively low up until $K$ reaches 5. After that, the error jumps up to around 25%. For $K = 8$, the training set error rate stays low up until $K > 8$. In general, the training set error rate is expected to be low for all values of $K \leq K'$, because the training algorithm is designed to store the $K'$ nearest neighbors in the training set important in performing the classification.

The performance on the testing set, however, does not show such behavior. All three error rate curves seem to stay between 20% and 30%. As $K'$ increases, there is a slight increase in the error rates of the classifiers. At $K = 8$, a CKNN classifier trained using $K' = 1$ provides an error rate of 21.0% while a CKNN classifier trained using $K' = 5$ gives an error rate of 23.1%, and one using $K' = 8$, an error rate of 33.0%. This reduction in performance is probably due to overfitting of the training set, which is evident in the error rate curves for the training set. In addition to reducing the
test set performance, increasing $K'$ also increases the number of training vectors that have to be stored. For these reasons, a value of one for the free parameter $K'$ was used for the CKNN classifier simulations.

4.6 KD Tree Classifier

4.6.1 Bucket Size

The KD tree classifier, as described in Section 2.5, has two free parameters, one is $K$, the number of nearest neighbors used in the voting procedure to perform the classification, and the other is the bucket size, the maximum number of data points in each terminal node or leaf of the binary splitting tree structure that is constructed from the training data. Since the KD tree classifier is guaranteed to find the $K$ nearest neighbors, and, as a result, will have the same performance as a conventional $K$ nearest neighbor classifier, the error rate of the classifier is not affected by the value of the bucket size. Instead, only the size of the tree and the efficiency of the search for the $K$ nearest neighbors is affected by the bucket size.

Given a set of training data with $N$ vectors, the smaller the bucket size, the larger the KD tree structure will have to be in order to be able to hold all the vectors in the training set. In the extreme case where the bucket size is equal to one, there will be as many nodes as there are data vectors. At the other extreme where the bucket size is greater than or equal to $N$, there will be only one node in the tree. With just one node holding all the training vectors, the KD tree essentially reduces to the conventional $K$ nearest neighbor classifier.

Figure 4.27 shows two plots of the performance of the KD tree classifier on the disjoint problem. Plot (A) shows the number of stored exemplars examined out of a total of 500 and Plot (B) shows the amount of CPU time required by the KD tree classifier as a function of the number of nearest neighbors, $K$, for a range of different bucket sizes. The CPU times shown in Figure 4.27(B) are the times needed by the KD tree algorithm to build and search a tree for the $K$ nearest neighbors. The times were measured on a Sun 3/110 workstation with a floating point accelerator. In general,
4.6. **KD TREE CLASSIFIER**

![Diagram](image)

Figure 4.27: (A) Number of Exemplars Examined and (B) Time Required by the KD Tree Classifier for the Disjoint Problem as a Function of the Bucket Size.

searching for more neighbors (ie. having a larger $K$) requires more time and requires examining more stored exemplars. As evident from Figure 4.27, these increases are almost linear with respect to the number of nearest neighbors.

Making the bucket size smaller reduces the number of stored exemplars that have to be search as evidenced by the curves in Figure 4.27(A), but does not necessarily reduce the amount of time needed to perform the search. In fact, from Figure 4.27(B), a bucket size of one requires the most time. This is because a check to see whether any other buckets overlap the current radius from the unknown vector to its nearest neighbors must be performed for each bucket. With only one training vector in each bucket, this check must be made for each training vector close to the unknown vector near the bottom of the tree. With several vectors in each bucket, this check needs to be performed only once for each bucket.

The advantage of having a small bucket size is that very few “extra” distance calculations have to be performed every time a terminal node is reached because there are very few data vectors in each bucket. The disadvantage is that a lot more buckets may have to be searched, which means that a lot more overhead may be
required to perform the checks and to move around the tree. Increasing the bucket size increases the number of exemplars that have to be searched (i.e. distance calculations performed) every time a terminal node is reached, but reduces the number of buckets that have to be searched, and hence, the number of checks that have to be performed.

There is a tradeoff between computing the extra distances and making the overhead calculations. This tradeoff can be seen in Figure 4.27(B). With a bucket size of one, a lot of time is needed in order to perform the search, due to the extra overhead calculations. Increasing the bucket size to two reduces the amount of time required, and increasing it further, to five, reduces the amount of time required even more. In these cases, the cost of computing the extra two or three distances is less than the cost of performing the extra checks in the bucket size equal to one case. Further increases in the bucket size, however, result in an increase in the amount of time required, due probably to the increased number of “extra” distance calculations. All the problems investigated in this study exhibited this behavior. For small and large bucket sizes, the computational time was increased, but for intermediate bucket sizes, the computational time was the lowest. For the remaining simulations using the KD tree classifier, a bucket size of five was used.

4.6.2 Number of Nearest Neighbors

For the standard $K$ nearest neighbor (KNN) classifier and the KD tree implementation of the KNN classifier, the training set error can be minimized, in fact set to zero, by selecting the number of nearest neighbors, $K = 1$. This value of the free parameter $K$, however, often does not minimize the test set error. The more dissimilar the training set is from the test set, the greater the divergence between the training set error rate and the test set error rate. This divergence can be seen in Figure 4.28. Plot (A), on the left, shows the error rate of the KD tree classifier on the vowel problem as a function of the number of nearest neighbors. For $K = 1$, the training set error rate is zero, but the testing set error rate is very high, at around 25%. It is not until $K$ is greater than 6, that the test set error rate begins to reach its minimum. For problems where the training set is a good representation of the unknown testing set, choosing $K$ to minimize the training set error rate will give good
results. For example, Figure 4.28(B) shows the error rate of the KD tree classifier on the disjoint problem as a function of the number of nearest neighbors. In this case, the value of \( K \) that minimizes the training set error rate also minimizes the test set error rate.

4.7 Chapter Summary

In this chapter, experiments were described that were performed to find appropriate values for the free parameters and to determine the proper complexity of the classifiers. For many of the classifiers, the range of different values of free parameters had to be searched in order to find a set of values which provided good classification performance. In addition, many classifiers showed a divergence in performance on the training and test sets as the complexity of the classifier increased. In these cases, cross validation and statistical techniques were used to determine proper classifier size. For problems where the training and test set performances follow each other closely, the classifier size was determined by selecting the complexity required to perform well on the training set.
Chapter 5

Classifier Comparison and Discussion

The six different classifiers (radial basis function, high order polynomial (GMDH) network, linear decision tree, Gaussian mixture, KD-tree, and condensed $K$ nearest neighbor classifiers) described in Chapter 4 were tested on the four sets of data (bullseye, disjoint, vowel, and digit) described in Chapter 3. This chapter presents a comparison of the performance of these classifiers on these problems, in terms of

1. Error Rate,

2. Type of Decision Regions Formed,

3. Training and Classification Memory Usage, and

4. Training and Classification Times.

In comparing the performance of the different classifiers, one high performing version of each of the classifiers, as determined from the tuning and complexity experiments described in Chapter 4, was selected as a representative for each problem. Table 5.1 shows the values of the free parameters and the size and structure for each of the selected classifiers used in the comparisons.

A summary of the performance of each of the classifiers on the four different problems in terms of error rates, training and classification times, and training and classification memory usages is given in Table 5.2. Error rate is given in percentage error, training time is given in CPU seconds, classification time is given in CPU milliseconds, and memory usage is given in bytes.
### Table 5.1: Summary of Parameter Values and Sizes for Selected Classifiers.

<table>
<thead>
<tr>
<th>Problem</th>
<th>RBF</th>
<th>RBF-MS</th>
<th>L-TREE</th>
<th>GMDH-CV</th>
<th>GMDH-PSE</th>
<th>GMIX</th>
<th>KD-TREE</th>
<th>C-KNN</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Bullseye</strong></td>
<td>30 Basis Functions, K-means Centers, $H = 20.0$, Diagonal Covariance</td>
<td>30 Centers, 80 Basis Functions</td>
<td>27 Nodes Before Pruning, 17 After Pruning</td>
<td>8 Layers, Maximum Width of 20 (4-fold Cross Validation)</td>
<td>10 Layers, Maximum Width of 4, $\sigma_p^2 = 0.012$</td>
<td>Normal-Diagonal Covariance (1 per component), 1 Component per Class</td>
<td>Bucket Size of 5, $K = 1$</td>
<td>35 Stored Exemplars, $K = 1$</td>
</tr>
<tr>
<td><strong>Disjoint</strong></td>
<td>30 Basis Functions, K-means Centers, $H = 3.0$, Diagonal Covariance</td>
<td>45 Centers, 143 Basis Functions</td>
<td>35 Nodes Before Pruning, 29 After Pruning</td>
<td>7 Layers, Maximum Width of 45 (4-fold Cross Validation)</td>
<td>7 Layers, Maximum Width of 18, $\sigma_p^2 = 0.032$</td>
<td>Normal-Diagonal Covariance (1 per component), 16 Components per Class</td>
<td>Bucket Size of 5, $K = 2$</td>
<td>49 Stored Exemplars, $K = 1$</td>
</tr>
<tr>
<td><strong>Vowel</strong></td>
<td>30 Basis Functions, K-means Centers, $H = 0.8$, Diagonal Covariance</td>
<td>45 Centers, 77 Basis Functions</td>
<td>161 Nodes Before Pruning, 27 After Pruning</td>
<td>4 to 7 Layers, Maximum Width of 8 (4-fold Cross Validation)</td>
<td>3 to 8 Layers, Maximum Width of 16, $\sigma_p^2 = 0.05$</td>
<td>Normal-Diagonal Covariance (1 per component), 2 Components per Class</td>
<td>Bucket Size of 5, $K = 8$</td>
<td>152 Stored Exemplars, $K = 8$</td>
</tr>
<tr>
<td><strong>Digit</strong></td>
<td>30 Basis Functions, K-means Centers, $H = 20.0$, Diagonal Covariance</td>
<td>30 Centers, 90 Basis Functions</td>
<td>54.4 Nodes Before Pruning, 29.5 After Pruning (Average over 16 Talkers)</td>
<td>1 to 2 Layers, Maximum Width of 25, $\sigma_p^2 = 1.0$</td>
<td>3 to 5 Layers, Maximum Width of 25, (2-fold Cross Validation)</td>
<td>Diagonal Across-Class-Grand Covariance (1 matrix), 2 Components per Class</td>
<td>Bucket Size of 5, $K = 2$</td>
<td>Average of 15 Stored Exemplars Across the 16 Talkers, $K = 1$</td>
</tr>
<tr>
<td></td>
<td>MEMORY TRAINING (bytes)</td>
<td>MEMORY TESTING (bytes)</td>
<td>TIME TRAINING (cpu sec)</td>
<td>TIME TESTING (cpu msec)</td>
<td>ERROR TRAINING (%)</td>
<td>ERROR TESTING (%)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>---------------------</td>
<td>-------------------------</td>
<td>------------------------</td>
<td>-------------------------</td>
<td>-------------------------</td>
<td>-------------------</td>
<td>------------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Bullseye Problem</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RBF</td>
<td>4560</td>
<td>600</td>
<td>51.4</td>
<td>6.4</td>
<td>0.4</td>
<td>0.8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RBF-MS</td>
<td>28160</td>
<td>1600</td>
<td>271.4</td>
<td>14.4</td>
<td>0.6</td>
<td>0.4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>L-TREE</td>
<td>10832</td>
<td>476</td>
<td>23.3</td>
<td>0.24</td>
<td>1.0</td>
<td>1.6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GMDH-CV</td>
<td>89760</td>
<td>2295</td>
<td>281.7</td>
<td>10.2</td>
<td>1.4</td>
<td>1.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GMDH-PSE</td>
<td>19040</td>
<td>840</td>
<td>27.0</td>
<td>5.0</td>
<td>0.0</td>
<td>0.6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GMIX</td>
<td>80</td>
<td>32</td>
<td>1.7</td>
<td>7.2</td>
<td>1.4</td>
<td>1.2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>KD-TREE</td>
<td>8040</td>
<td>8040</td>
<td>1.9</td>
<td>6.2</td>
<td>0.0</td>
<td>1.8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C-KNN</td>
<td>420</td>
<td>420</td>
<td>1.7</td>
<td>5.6</td>
<td>0.0</td>
<td>2.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Disjoint Problem</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RBF</td>
<td>4560</td>
<td>600</td>
<td>43.3</td>
<td>6.4</td>
<td>1.4</td>
<td>2.8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RBF-MS</td>
<td>86372</td>
<td>2860</td>
<td>563.9</td>
<td>20.6</td>
<td>1.0</td>
<td>2.6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>L-TREE</td>
<td>11664</td>
<td>812</td>
<td>25.0</td>
<td>0.23</td>
<td>0.4</td>
<td>2.2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GMDH-CV</td>
<td>198440</td>
<td>2016</td>
<td>1107.5</td>
<td>15.2</td>
<td>3.65</td>
<td>4.7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GMDH-PSE</td>
<td>79856</td>
<td>952</td>
<td>264.3</td>
<td>8.0</td>
<td>1.4</td>
<td>2.8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GMIX</td>
<td>1280</td>
<td>512</td>
<td>64.1</td>
<td>15.6</td>
<td>0.8</td>
<td>2.8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>KD-TREE</td>
<td>8040</td>
<td>8040</td>
<td>1.9</td>
<td>6.4</td>
<td>0.0</td>
<td>2.8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C-KNN</td>
<td>588</td>
<td>588</td>
<td>3.0</td>
<td>6.6</td>
<td>0.0</td>
<td>2.4</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Vowel Problem</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RBF</td>
<td>6480</td>
<td>1560</td>
<td>24.9</td>
<td>9.6</td>
<td>20.8</td>
<td>20.5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RBF-MS</td>
<td>31108</td>
<td>4004</td>
<td>146.7</td>
<td>18.9</td>
<td>21.0</td>
<td>19.5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>L-TREE</td>
<td>22176</td>
<td>756</td>
<td>25.4</td>
<td>0.33</td>
<td>18.3</td>
<td>21.9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GMDH-CV</td>
<td>251648</td>
<td>9296</td>
<td>284.1</td>
<td>31.6</td>
<td>27.8</td>
<td>27.7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GMDH-PSE</td>
<td>482760</td>
<td>4256</td>
<td>965.2</td>
<td>39.9</td>
<td>20.7</td>
<td>21.9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GMIX</td>
<td>800</td>
<td>320</td>
<td>10.0</td>
<td>14.4</td>
<td>20.7</td>
<td>20.7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>KD-TREE</td>
<td>5408</td>
<td>5408</td>
<td>1.3</td>
<td>18.0</td>
<td>20.4</td>
<td>17.4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C-KNN</td>
<td>1824</td>
<td>1824</td>
<td>16.4</td>
<td>28.1</td>
<td>26.3</td>
<td>21.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Digit Problem</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RBF</td>
<td>8160</td>
<td>3600</td>
<td>17.6</td>
<td>44.6</td>
<td>0.53</td>
<td>5.4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RBF-MS</td>
<td>46080</td>
<td>10800</td>
<td>71.3</td>
<td>121.4</td>
<td>0.0</td>
<td>8.9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>L-TREE</td>
<td>21346</td>
<td>826</td>
<td>62.4</td>
<td>2.3</td>
<td>13.5</td>
<td>56.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GMDH-CV</td>
<td>149660</td>
<td>4256</td>
<td>806.4</td>
<td>85.7</td>
<td>1.9</td>
<td>25.5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GMDH-PSE</td>
<td>117320</td>
<td>1176</td>
<td>676.9</td>
<td>71.4</td>
<td>2.4</td>
<td>16.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GMIX</td>
<td>2880</td>
<td>1408</td>
<td>7.3</td>
<td>66.0</td>
<td>1.1</td>
<td>6.7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>KD-TREE</td>
<td>6688</td>
<td>6688</td>
<td>1.9</td>
<td>56.2</td>
<td>2.0</td>
<td>7.6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C-KNN</td>
<td>1380</td>
<td>1380</td>
<td>4.9</td>
<td>33.0</td>
<td>0.18</td>
<td>9.4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5.2: Performance Summary of the Selected Classifiers.
5.1 Decision Regions and Feature Space Partitioning

The classifiers differ from each other not only in their structure and in how they are trained, but also in the type of decision boundaries and decision regions formed. Characteristics of the decision regions depend on the type of discriminant functions formed. For example, in the linear decision tree classifier, hyperplanes separate different classes and decision regions are hyper-polygons formed by the intersection of these hyperplane surfaces. For the radial basis function classifier, discriminant functions are formed by a linear combination of Gaussian pulses and decision boundaries are relatively smooth curves.

The type of decision region formed by a classifier plays an important role in determining whether the classifier can generalize well to unseen data or with high dimensionality inputs. In general, decision regions that have many irregularities or very fine structure have the potential to overfit the training data and result in poor performance on unseen test data. One of the main reasons for reducing the complexity of a classifier, as mentioned in Chapter 4, is to reduce the complexity and fine structure of the decision regions so that they can generalize well to the test set. This is especially true for classifiers like the linear decision tree classifier and the KD-tree classifier, which have the ability to correctly classify the entire training set.

The decision regions formed by the radial basis function classifier for the vowel problem are shown in Figure 5.1. Boundaries are smooth spline-like curves that can form arbitrarily complex regions. This makes the RBF classifier less susceptible to overfitting the training set and enables the RBF classifier to generalize well on unseen test data. The effect of the radial nature of the basis functions can be seen along the edges of the decision regions where curves look like circular arcs.
Figure 5.1: Decision Regions Formed by the RBF Classifier for the Vowel Problem.
Figure 5.2: Decision Regions Formed by the Gaussian Mixture Classifier for the Vowel Problem.

Decision boundaries formed by the Gaussian mixture classifier are also smooth spline-like curves as shown in Figure 5.2. These decisions regions, however, differ from those of the RBF classifier in that they are not bounded on the outside edges by arc-like curves. Instead, the decision regions extend out to regions where there is no data. Because of the smooth decision boundaries, the Gaussian mixture classifier has the potential, like the RBF classifier, to generalize well to unseen test data and to high dimensions.
The hyper-polygon decision regions formed by the linear decision tree classifier are shown in Figure 5.3. In two dimensions, the hyperplanes reduce to straight lines and the hyper-polygons reduce to 2D-polygons. Because the decision boundaries are not smoothly interpolated to lie in between data points, but are hyperplanes placed to try to completely classify the training set, the decision regions formed may be too specific to the training set. As a result, the classifier may not be able to generalize well to unseen test data. This is especially true if the training and test sets are dissimilar or if there is not enough training data to specify the decision regions accurately.
Figure 5.4: Decision Regions Formed by the GMDH Classifier for the Vowel Problem.

Decision regions for the high order polynomial (GMDH) network classifier, shown in Figure 5.4, are smooth and well behaved in regions of the input space that are densely sampled by the training data. In regions where there is no data, however, decision regions are erratic. This is because the high order polynomials are forced to be relatively smooth in the data rich regions and, as a result, oscillate wildly outside these regions in order to satisfy the order of the polynomial. For this reason, the high order polynomial network is not able to extrapolate well. Classifying a data point that lies outside the region of training can result in unexpected behavior.
Figure 5.5: Decision Regions Formed by the KD Tree Classifier for the Vowel Problem.

Decision regions formed by the KD-tree classifier resemble those formed by the RBF and Gaussian mixture classifiers, except boundaries have more irregularities and fine structure. This may lead to overfitting of the training data and poor generalization, especially if the training set is not a good representation of the test set.
5.2 Classification Error Rate

5.2.1 McNemar's Significance Test

In error rate comparisons, it is important to be able to determine whether any apparent difference in the performance of the classifiers is actually statistically significant. In order to determine this, McNemar's test, described by Gillick and Cox in [11], was used. This test applies to two classifiers at a time and gives a measure of significance to the observed differences in their performance on a given data set. A probability measure, $P$, of the null hypothesis, $H_0$, which assumes that the two classifiers perform equally well, is produced by this test. The hypothesis $H_0$ is accepted if $P$ is greater than some significance level, $\alpha$, meaning that the difference in performance between the two classifiers is not statistically significant. However, if $P$ is less than $\alpha$, then $H_0$ is rejected, meaning that the difference in performance between the two classifiers is statistically significant. Because McNemar's test requires that the errors made be independent events, this significance test is appropriate for comparing pattern classification algorithms, which classify individual data vectors independently of each other.

The performance of two classifiers, $C_1$ and $C_2$, on one set of data can be summarized in a $2 \times 2$ table format:

\[
\begin{array}{cc|cc}
 & C2 & \\
C1 & Correct & Incorrect & \\
Correct & N_{00} & N_{01} & \\
Incorrect & N_{10} & N_{11} & \\
\end{array}
\]

where
\[ N_{00} = \text{Number of patterns which C1 classifies correctly, and C2 classifies correctly} \]

\[ N_{01} = \text{Number of patterns which C1 classifies correctly, and C2 classifies incorrectly} \]

\[ N_{10} = \text{Number of patterns which C1 classifies incorrectly, and C2 classifies correctly} \]

\[ N_{11} = \text{Number of patterns which C1 classifies incorrectly, and C2 classifies incorrectly.} \]

The null hypothesis, \( H_o \), represents the assertion that, given that only one of the classifiers makes a mistake, it is equally likely to be either one. Thus, if \( q \) is the conditional probability that \( C1 \) will make an error given that either \( C1 \) or \( C2 \) makes an error, then \( H_o \) implies that \( q = \frac{1}{2} \).

In order to determine if hypothesis \( H_o \) is correct, it is necessary to examine only the patterns on which only one of the classifiers made a mistake, i.e. only those cases that contribute to \( N_{10} \) and \( N_{01} \). The cases where both classifiers are correct (\( N_{00} \)) or incorrect (\( N_{11} \)) do not provide any information about the relative performance of the classifiers. If there are \( k = N_{10} + N_{01} \) patterns where only one classifier makes an error and hypothesis \( H_o \) is true, then \( N_{10} \) will have a binomial, \( B(k, \frac{1}{2}) \), distribution.

The probability, \( P \), that \( H_o \) is correct can be computed directly from the definition of the Binomial distribution (equation 5.1) or can be approximated by a Normal distribution if \( k \) is large enough (\( k > 50 \)). This probability is reported in the remainder of this chapter. Note again that a high value of \( P \) near one means that the two classifiers have similar error rates, whereas a low value near zero means that the error rates are dissimilar.
\[ 2Pr(0 \leq m \leq N_{10}) = 2 \sum_{m=0}^{N_{10}} \binom{k}{m} \left( \frac{1}{2} \right)^k \quad \text{when } N_{10} < \frac{k}{2} \]

\[ P = \begin{cases} 
1.0 & \text{when } N_{10} = \frac{k}{2} \\
2Pr(N_{10} \leq m \leq k) = 2 \sum_{m=N_{10}}^{k} \binom{k}{m} \left( \frac{1}{2} \right)^k & \text{when } N_{10} > \frac{k}{2} 
\end{cases} \quad (5.1) \]

### 5.2.2 Test Set Error Rate

Figure 5.6 plots the classification (test set) error rates for the different classifiers on the bullseye, disjoint, vowel, and digit problems. The solid line in each plot represents the mean test set error rate across all the classifiers for that problem, and the dashed lines are one binomial standard deviation, \( \sigma \), away from the mean. The binomial standard deviation is calculated simply as

\[ \sigma = \sqrt{\frac{\mathcal{E}(1 - \mathcal{E})}{N}}, \quad (5.2) \]

where \( \mathcal{E} \) is the estimated average error rate across the classifiers for that problem, and \( N \) is the number of patterns in the test set for that problem. The standard deviation, \( \sigma \), gives a rough measure of the range of statistical fluctuation of the error rates of the different classifiers.

A more detailed statistical analysis of the test set error rate for all possible pairs of the different classifiers, using McNemar's test, is given for each of the four problems in Tables 5.3 through 5.6. These tables are arranged in a square 8×8 matrix, where each \( i, j \) entry in the table is the \( P \) value defined in equation 5.1 for the pair of classifiers in row \( i \) (classifier \( C1 \)), and column \( j \), (classifier \( C2 \)). For the comparisons in this chapter, the significance level, \( \alpha \), is set equal to 0.01. To make the significance tables easier to read, all the entries in the significance tables with \( P \) values less than or equal to \( \alpha = 0.01 \) are replaced by asterisks (*). Asterisks thus indicate classifiers
Figure 5.6: Test Set Error Rates of the Different Classifiers on the Four Problems.
Table 5.3: Statistical Significance ($P$) of Test Set Error Rate Differences for the Bullseye Problem (Asterisks Indicate Differences that are Statistically Significant at the 0.01 Level).

<table>
<thead>
<tr>
<th>Bullseye Problem</th>
<th>RBF MS</th>
<th>RBF PSE</th>
<th>GMDH CV</th>
<th>GMDH PSE</th>
<th>C-KNN</th>
<th>KD-TREE</th>
<th>L-TREE</th>
<th>GMIX</th>
</tr>
</thead>
<tbody>
<tr>
<td>RBF</td>
<td>—</td>
<td>0.5</td>
<td>1.0</td>
<td>0.69</td>
<td>0.11</td>
<td>0.12</td>
<td>0.29</td>
<td>0.75</td>
</tr>
<tr>
<td>RBF-MS</td>
<td>0.5</td>
<td>—</td>
<td>1.0</td>
<td>0.125</td>
<td>*</td>
<td>0.015</td>
<td>0.031</td>
<td>0.29</td>
</tr>
<tr>
<td>GMDH-PSE</td>
<td>1.0</td>
<td>1.0</td>
<td>—</td>
<td>0.375</td>
<td>0.065</td>
<td>0.07</td>
<td>0.18</td>
<td>0.51</td>
</tr>
<tr>
<td>GMDH-CV</td>
<td>0.68</td>
<td>0.12</td>
<td>0.37</td>
<td>—</td>
<td>0.39</td>
<td>0.45</td>
<td>0.73</td>
<td>1.0</td>
</tr>
<tr>
<td>C-KNN</td>
<td>0.11</td>
<td>*</td>
<td>0.065</td>
<td>0.39</td>
<td>—</td>
<td>1.0</td>
<td>0.79</td>
<td>0.45</td>
</tr>
<tr>
<td>KD-TREE</td>
<td>0.12</td>
<td>0.015</td>
<td>0.07</td>
<td>0.45</td>
<td>1.0</td>
<td>—</td>
<td>1.0</td>
<td>0.61</td>
</tr>
<tr>
<td>L-TREE</td>
<td>0.29</td>
<td>0.031</td>
<td>0.18</td>
<td>0.73</td>
<td>0.79</td>
<td>1.0</td>
<td>—</td>
<td>0.79</td>
</tr>
<tr>
<td>GMIX</td>
<td>0.75</td>
<td>0.29</td>
<td>0.51</td>
<td>1.0</td>
<td>0.45</td>
<td>0.61</td>
<td>0.79</td>
<td>—</td>
</tr>
</tbody>
</table>

With the exception of the high order polynomial GMDH network classifier using the regularity error criterion, labeled as “GMDH-CV”, the error rates of the different classifiers on the bullseye, disjoint, and vowel problems do not differ significantly from each other. There is one small exception to this. Table 5.3 indicates that for the bullseye problem, the performance difference between the multi-scale radial basis function (RBF-MS) classifier and the condensed $K$ nearest neighbor (CKNN) classifier is statistically significant. This one exception, however, can be attributed to the exceptionally good performance of the multi-scale RBF classifier on the bullseye problem.

Although the high order polynomial network GMDH-CV classifier performs poorly on the disjoint and vowel problems, the high order polynomial network GMDH-PSE classifier, which uses the predicted squared error criterion, is able to perform well on both of these problems. As discussed in Section 4.2, since both versions of the GMDH network classifier have the same set of polynomial functions to work from, the discrepancy in performance is due entirely to the different approach used to determine network complexity. Previously, the regularity error criterion was shown to be poor in selecting a good set of polynomials and also poor in estimating model parameters.
## 5.2. CLASSIFICATION ERROR RATE

<table>
<thead>
<tr>
<th>Disjoint Problem</th>
<th>RBF</th>
<th>RBF MS</th>
<th>GMDH PSE</th>
<th>GMDH CV</th>
<th>C-KNN</th>
<th>KD-TREE</th>
<th>L-TREE</th>
<th>GMIX</th>
</tr>
</thead>
<tbody>
<tr>
<td>RBF</td>
<td>—</td>
<td>1.0</td>
<td>1.0</td>
<td>0.10</td>
<td>0.79</td>
<td>1.0</td>
<td>0.65</td>
<td>1.0</td>
</tr>
<tr>
<td>RBF-MS</td>
<td>1.0</td>
<td>—</td>
<td>1.0</td>
<td>0.052</td>
<td>1.0</td>
<td>1.0</td>
<td>0.79</td>
<td>1.0</td>
</tr>
<tr>
<td>GMDH-PSE</td>
<td>1.0</td>
<td>1.0</td>
<td>—</td>
<td>0.063</td>
<td>0.77</td>
<td>1.0</td>
<td>0.63</td>
<td>1.0</td>
</tr>
<tr>
<td>GMDH-CV</td>
<td>0.10</td>
<td>0.052</td>
<td>0.063</td>
<td>—</td>
<td>0.026</td>
<td>0.093</td>
<td>0.017</td>
<td>0.13</td>
</tr>
<tr>
<td>C-KNN</td>
<td>0.79</td>
<td>1.0</td>
<td>0.77</td>
<td>0.026</td>
<td>—</td>
<td>0.75</td>
<td>1.0</td>
<td>0.84</td>
</tr>
<tr>
<td>KD-TREE</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>0.093</td>
<td>0.75</td>
<td>—</td>
<td>0.61</td>
<td>1.0</td>
</tr>
<tr>
<td>L-TREE</td>
<td>0.65</td>
<td>0.79</td>
<td>0.63</td>
<td>0.017</td>
<td>1.0</td>
<td>0.6</td>
<td>—</td>
<td>0.70</td>
</tr>
<tr>
<td>GMIX</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>0.18</td>
<td>0.84</td>
<td>1.0</td>
<td>0.70</td>
<td>—</td>
</tr>
</tbody>
</table>

Table 5.4: Statistical Significance ($P$) of Test Set Error Rate Differences for the Disjoint Problem.

<table>
<thead>
<tr>
<th>Vowel Problem</th>
<th>RBF</th>
<th>RBF MS</th>
<th>GMDH PSE</th>
<th>GMDH CV</th>
<th>C-KNN</th>
<th>KD-TREE</th>
<th>L-TREE</th>
<th>GMIX</th>
</tr>
</thead>
<tbody>
<tr>
<td>RBF</td>
<td>—</td>
<td>0.61</td>
<td>0.63</td>
<td>*</td>
<td>0.89</td>
<td>0.11</td>
<td>0.53</td>
<td>1.0</td>
</tr>
<tr>
<td>RBF-MS</td>
<td>0.61</td>
<td>—</td>
<td>0.26</td>
<td>*</td>
<td>0.59</td>
<td>0.28</td>
<td>0.27</td>
<td>0.57</td>
</tr>
<tr>
<td>GMDH-PSE</td>
<td>0.63</td>
<td>0.26</td>
<td>—</td>
<td>*</td>
<td>0.90</td>
<td>0.038</td>
<td>1.0</td>
<td>0.76</td>
</tr>
<tr>
<td>GMDH-CV</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>—</td>
<td>*</td>
<td>0.023</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>C-KNN</td>
<td>0.89</td>
<td>0.59</td>
<td>0.90</td>
<td>*</td>
<td>—</td>
<td>0.11</td>
<td>0.79</td>
<td>1.0</td>
</tr>
<tr>
<td>KD-TREE</td>
<td>0.11</td>
<td>0.28</td>
<td>0.038</td>
<td>*</td>
<td>0.11</td>
<td>—</td>
<td>0.035</td>
<td>0.1</td>
</tr>
<tr>
<td>L-TREE</td>
<td>0.53</td>
<td>0.27</td>
<td>1.0</td>
<td>0.023</td>
<td>0.79</td>
<td>0.035</td>
<td>—</td>
<td>.69</td>
</tr>
<tr>
<td>GMIX</td>
<td>1.0</td>
<td>0.57</td>
<td>0.76</td>
<td>*</td>
<td>1.0</td>
<td>0.1</td>
<td>0.69</td>
<td>—</td>
</tr>
</tbody>
</table>

Table 5.5: Statistical Significance ($P$) of Test Set Error Rate Differences for the Vowel Problem (Asterisks Indicate Differences that are Statistically Significant at the 0.01 Level).
because of the partitioning of the training set into estimation and evaluation sets. This inability to select and form good polynomials is reflected in the poor performance of the GMDH-CV classifier. Attempts to increase the amount of data used in the evaluation and estimation process by first determining the size of the GMDH network classifier using cross-validation and then using all the training data to estimate and evaluate the polynomials did not work. As discussed in Section 4.2, this resulted in classifiers that overfit the training data and did not perform well on the unseen test data.

Performance on the digit problem, unlike on the other problems, varies widely and differs significantly from classifier to classifier, as evident from Figure 5.6 and Table 5.6. The digit problem is more difficult than the other three problems. It has very little training data and high dimensional inputs. There are only 70 training patterns per speaker for the 7 different classes, which means that there are only 10 patterns for each class. The input dimension, $D$, of the pattern vectors is 22, whereas in all the other problems, $D$ is equal to 2. Some classifiers, like the RBF and Gaussian mixture classifiers, are able to achieve very low error rates on this problem even with the high dimensionality and small training set size. Other classifiers, like the multi-scale RBF, KD-tree, and condensed $K$ nearest neighbor classifiers, however, are only able to achieve intermediate error rates. The high order polynomial GMDH network
5.2. **CLASSIFICATION ERROR RATE**

Classifiers and the linear decision tree classifier are not able to perform well on this problem at all.

The linear tree classifier performs poorly because there is not enough training data to sample the input space densely enough for the training algorithm to form decision boundaries that can generalize well. The actual decision boundaries are not adequately defined by the limited amount of training data, which means that any decision boundaries constructed using this data will probably overfit the data. This overfitting results in good performance on the training set, but poor performance on the test set. Given more training data, however, the input space may be better sampled and the resulting decision regions will probably be able to generalize better.

The poor performance of the GMDH network classifiers on the digit problem is due, in part, to the inability of the GMDH network classifier to extrapolate to regions where there is very little or no training data. This behavior can be seen in the decision region plot of Figure 5.4 in Section 5.1. The boundaries formed by the GMDH network classifier are smooth in regions where there is data, but outside these regions, the decision boundaries are erratic. In the digit problem, where there is little training data, the entire input space spanned by the vectors in the test set may not be adequately covered by the vectors in the training set. Because of this, the GMDH network classifier decision regions may not be properly trained (i.e. may not be smooth and well behaved) in the regions where there is test data, but no training data. Again, one solution to this problem is to provide more training data.

The RBF and Gaussian mixture classifiers perform well on the digit problem. The ability of these classifiers to perform well with high dimensionality and small training set size is probably due to the type of decision regions formed. As shown in Figures 5.1 and 5.2, the decision regions formed by these two classifiers are relatively smooth and well behaved. There are no "sharp" edges as in the case of the linear tree classifier or erratic behavior in the regions where there is no training data, as in the case of the GMDH network classifiers. These properties combine to give the RBF and Gaussian mixture classifiers the ability to generalize well to unseen data.

As discussed in Section 4.1, the multi-scale RBF classifier can not perform as well as the conventional RBF classifier on the digit problem because of the extra
parameters resulting from the extra basis function nodes. Reducing the number of parameters by pruning away some of the basis function nodes improved the performance, as shown previously in Figure 4.8, but not to the extent that performance equaled that of the conventional RBF classifier.

The KD tree and condensed $K$ nearest neighbor classifiers are also able to perform well on the digit problem. Decision regions formed by the nearest neighbor classifiers are similar to those of the RBF and Gaussian mixture classifiers, but are not as smooth. As a result, there is more potential in these cases to form decision regions that are too specific to the training data. Overall, classifiers that form smooth decision boundaries tend to generalize well, while those that form convoluted decision boundaries do not.

A few more observations can be made from the entries of the significance tables. In particular, across the different problems, the performance of the condensed $K$ nearest neighbor (CKNN) classifier does not differ significantly from the performance of the KD tree classifier, even though the CKNN classifier stores many fewer exemplar patterns than the KD tree classifier. In addition, the performance of the multi-scale RBF classifier, aside from the digit problem, does not differ significantly from the performance of the conventional RBF classifier.

5.2.3 Training Set Error Rate

Figure 5.7 plots the training set error rates for the different classifiers on the bullseye, disjoint, vowel, and digit problems. The solid line in each plot represents the mean training set error rate across all the classifiers for that problem, while the dashed lines are one binomial standard deviation away from the mean.

In general, the error rates on the training sets are lower than the error rates on the test sets. This is expected since the classifiers are trained and construct decision regions based on the training set. For the digit problem, the error rates of the classifiers are all very close to zero, implying that the classifiers are overfitting the training data. Most notable are the very low error rates for the GMDH network classifiers, indicating that the decision regions formed by the GMDH network classifier are highly specific. The GMDH network classifier performs well on the training set,
Figure 5.7: Training Set Error Rates of the Different Classifiers on the Four Problems.
but, as previously discussed, it does not perform well on the test set.

A few of the classifiers have the ability to correctly classify the entire training set. These include the KD tree, condensed $K$ nearest neighbor, and linear decision tree classifiers. As shown above and in Chapter 4, however, success in classifying the training set is not indicative of classification error on the test set or how well the classifier can generalize.

## 5.3 Memory Usage

### 5.3.1 Training Memory Usage
5.3. MEMORY USAGE

Training memory usage is the amount of memory required during the training process of the classifier, in addition to, but not including, the training algorithm. In other words, it is the amount of memory required to keep track of all the data structures needed by the training algorithm. For example, in the radial basis function classifier, the training algorithm requires the means and variances of the basis functions, the weights connecting the basis function nodes to the output nodes, the correlation matrix \( \mathbf{R} \), and the output matrix \( \mathbf{B} \) to be stored. In the linear decision tree classifier, the information required to define each node of the tree has to be stored. In the nearest neighbor classifiers, all exemplar patterns have to be stored. For all the classifiers, the memory usage is measured in bytes and all variables are either 4 byte integers, 4 byte floating point numbers, or 4 byte address pointers.

Figure 5.8 shows the amount of training memory required by the different classifiers for the four problems. The data for the standard \( K \) nearest neighbor (KNN) classifier is added for comparison purposes. Of all the classifiers, the condensed \( K \) nearest neighbor (CKNN) and Gaussian mixture classifiers use the least amount of memory. In the CKNN classifier case, this low memory requirement is due to the reduced number of training exemplars that have to be stored. In the Gaussian mixture classifier case, it is due to the small number of model parameters needed to adequately model the different class distributions.

Because the KD tree and KNN classifiers have to store all the patterns in the training set, they require intermediate amounts of memory during training. In addition, the RBF classifier, which has to store weight values, basis function parameters, and several large matrices, also requires intermediate amounts of memory. Typically these classifiers require 5 to 10 times more memory than the CKNN or Gaussian mixture classifiers.

The multiscale RBF classifier requires more memory than the conventional RBF classifier because of the extra basis functions used to span the range of different basis function widths. The linear decision tree classifier requires a large amount of memory during training because it has to construct and store an overly complex tree, which is later pruned back for classification. Although the training memory requirement for the linear tree classifier is large, the amount of classification memory once the tree is
pruned back is usually very modest.

The high order polynomial GMDH network classifiers require an enormous amount of memory during training. On the average, the amount of memory needed is 10 to 20 times that of the RBF classifier, or 200 times that of the Gaussian mixture classifier. Large memory requirements result from two requirements imposed by the training algorithm. One is that all polynomials in the lower layers that combine to form polynomials in the higher layers have to be stored until the network stops growing. Only when the network has stopped growing can the algorithm back trace down the network, marking the specific polynomials that contribute to the final output polynomial. In general, the number of polynomials used in the final output is much smaller than the number that needs to be stored. The other requirement is that new observations have to be computed at every layer of the network. Since these observations are derived from applying the polynomial functions at each layer to the vectors of the training set, this means that all the vectors in the training set have to be stored.

5.3.2 Classification Memory Usage

Classification memory is the amount of memory required by a classifier to hold all the information necessary in order to perform classification. Essentially it is all the information needed to generate the decision boundaries. For example, in the nearest neighbor classifiers, all that is needed to perform classification is the set of stored exemplar patterns. In the case of the GMDH network classifiers, it is the set of final output polynomials, one for each class. In the case of the RBF classifiers, it is the set of basis function parameters and the set of trained weights.

Figure 5.9 shows the amount of classification memory required by the different classifiers for the four problems. Once again, the data for the standard $K$ nearest neighbor (KNN) classifier is added for comparison purposes. For most classifiers, classification memory usage is much less than training memory usage. This behavior is expected since the training process usually involves extra overhead storage that is not needed in the final classifier. This is not true, however, for the nearest neighbor classifiers (KNN, CKNN, KD tree), which store exemplar patterns from the training
Figure 5.9: Classification Memory Usage for the Different Classifiers on the Four Problems.
set. In these cases, the training and classification memory requirements are exactly the same. In terms of training memory usage, the nearest neighbor classifiers have only intermediate memory requirements. In terms of classification memory usage, however, the same nearest neighbor classifiers have the highest memory requirements.

The GMDH network classifiers, which have large training memory requirements have only intermediate classification memory requirements. After the large structures built by these classifiers during training are reduced, the classifiers become memory efficient. The linear tree classifier also exhibits this behavior. After the complex tree structure built during training is pruned back, the linear tree classifier is among the most memory efficient. The CKNN and Gaussian mixture classifiers are also very memory efficient.

In general, the memory requirement in most classifiers is proportional to the input dimension and is thus typically greatest for the digit problem. This is especially true of classifiers that have to store parameters that depend on the dimensionality of the input. These include the RBF and Gaussian mixture classifiers. In addition, the memory requirement also seems to be proportional to the complexity of decision regions. For example, in the GMDH network classifiers, more complex polynomials are constructed for the 10 class vowel problem than for the 2 class bullseye problem. As a result, more memory is required for the vowel problem than the bullseye problem.

5.4 Training and Classification Time

The amount of time required for classification and training were measured on a Sun 3/110 workstation with a floating point accelerator. Training times were measured in CPU seconds, while classification times were measured in CPU milliseconds. Both training and classification times were measured using the time command provided in the Unix operating system. The classification time, in CPU milliseconds, for each pattern was obtained by dividing the total CPU time needed to classify the entire test set by the number of patterns in the test set. The training time for the RBF classifiers includes the time required for the K-means clustering used to determine the basis function centers.
Figure 5.10: Training Times in CPU Seconds for the Different Classifiers.
Figure 5.10 shows the training time for the different classifiers on the four data sets. The $K$ nearest neighbor (KNN) classifier stands out distinctively as the most rapidly trained classifier. Training a conventional KNN classifier amounts to storing the vectors in the training set as exemplar patterns. The other two nearest neighbor classifiers (KD tree and condensed $K$ nearest neighbor (CKNN)) require more training than the KNN classifier, but still train very rapidly. In the KD tree classifier, a simple tree structure is built into the stored training data. In the CKNN classifier, each vector in the training set is classified in order to determine whether or not it should be stored as an exemplar. Although the CKNN classifier requires more training time than either the KNN or the KD tree classifier, the resulting CKNN classifier has much lower memory requirements, and slightly faster classification times.

The Gaussian mixture classifier requires about as much time to train as the CKNN classifier. The linear tree classifier requires longer training times because it has to construct a complex tree structure. As discussed in Section 5.3, however, when this overly complex tree is pruned back, the resulting classifier is very memory and computationally efficient. The RBF classifier takes about the same amount of time to train as the linear tree classifier.

Of all the classifiers, the neural network multiscale RBF and GMDH classifiers require the most time to train. In the case of the multiscale RBF classifier, having the extra basis functions and computing the Gaussian outputs is computationally very expensive. Depending on the problem, the multiscale RBF classifiers are between 5 and 10 times slower than the conventional RBF classifier. The GMDH network classifiers take a long time to train because of all the coefficient estimations and polynomial evaluations that are required at each layer. On average, the GMDH network classifiers take 10 times longer to train than RBF classifiers, and 100 times longer than KD tree classifiers.

The classification time for all the different classifiers on the four data sets is shown in Figure 5.11. All the classifiers require about the same amount of time to perform classification, with the exception of the linear decision tree classifier, which is orders of magnitude faster than any of the other classifiers. The linear tree classifier requires only a few numerical comparisons at each node of the tree, whereas the other classifiers
Figure 5.11: Classification Times in CPU Milliseconds for the Different Classifiers.
require much more computationally expensive operations like exponentials or distance calculations. It is interesting to note that the KD tree and CKNN classifiers require about the same amount of time to classify an unknown input vector even though one stores all the training patterns and the other stores only a very small subset of the training patterns. For low dimensionality problems, both the KD tree and CKNN classifiers provide faster classification times than the conventional KNN classifier. For high dimensionality problems, however, there is no significant difference between the three classifiers. In fact, the overhead calculations required by the KD tree classifier may even result in slower classification times than the conventional KNN classifier. Across all the different classifiers, the high dimensional digit problem appears to consistently require more classification time than the other three low dimensional problems.

It is important to note that these training and classification times are based on implementations on a traditional single processor serial computer. Different classifier implementations, such as on parallel computers or in VLSI hardware, may have different results.

5.5 Trade-Offs Between Performance Criteria

As discussed in the previous two sections, there are trade-offs among the classifiers between different performance issues such as training time, classification time, and memory usage. In this section, data from Table 5.2 and Figures 5.11, 5.10, and 5.9 are replotted on a time versus memory scale to make the relationships between the different performance factors more explicit. In particular, the relationship between classification time and classification memory, and between training time and classification memory are examined.

5.5.1 Classification Time vs. Classification Memory

Figure 5.12 shows the relationship between classification time and classification memory usage of the different classifiers on the vowel problem. In general, the trend is for
Figure 5.12: Classification Time versus Classification Memory Usage for the Vowel Problem.
classifiers that have more classification memory to require more classification time.

It is interesting to note the relative relationships of the three different nearest neighbor classifiers for the vowel problem. The condensed $K$ nearest neighbor (CKNN) classifier is slightly faster and requires less classification memory than the conventional KNN classifier because of the reduced number of patterns that have to be stored. The KD tree classifier, on the other hand, is much faster than the conventional KNN classifier, but requires slightly more memory because of the tree structure built around the stored training set that enables it to perform fast searches.

As noted before, the high order polynomial GMDH network classifiers require the most classification time and memory. At the other extreme, the linear tree classifier requires very little memory and classifies almost instantaneously. The Gaussian mixture and RBF classifiers show relatively low memory requirements and classification times. Due to the extra basis functions, the multiscale RBF classifier requires more time and memory than the conventional RBF classifier.

5.5.2 Training Time vs. Classification Memory

Figure 5.13 shows the relationship between training time and classification memory usage of the different classifiers on the vowel problem. With the exception of the GMDH network classifiers, the general trend is for classifiers that have low classification memory requirements to have long training time requirements. This behavior can be expected if the training process is viewed as a compression of the decision boundary information contained in the training data. The more compact the representation, the more time the processing will require.

The KNN and KD-tree classifiers are the most rapidly trained, but require large amounts of classification memory. As discussed before, these classifiers store all training vectors as exemplar patterns. The Gaussian mixture and linear decision tree classifiers require more training time, but use very little classification memory. For the vowel problem, the Gaussian mixture classifier requires 20 times less memory than either the KD tree or KNN classifiers. The RBF and CKNN classifiers require intermediate classification memory and training times. It is important to note that in general, trade-offs in performance characteristics depend on the particular problem
Figure 5.13: Training Time versus Classification Memory Usage for the Vowel Problem.
and can also vary for different implementations of the classifiers.

5.6 Chapter Summary

In this chapter, the six different classifiers were compared on the bullseye, disjoint, vowel, and digit problems in terms of error rate, type of decision regions formed, training and classification memory usage, and training and classification time.

On the bullseye, disjoint, and vowel problems, the classification error rate was similar across all the classifiers, except with the GMDH classifier using the regularity error criterion. On the digit problem, the combination of high dimensionality and little training data caused the GMDH and linear decision tree classifiers to construct classifiers that overfit the training data. Some of the other classifiers (the RBF and Gaussian mixture classifiers) were able to handle the high dimensionality and small training set size and provide low error rates.

Training algorithms and the resulting decision regions formed by each classifier were very different. The characteristics of the decision regions formed by a classifier played an important role in determining generalization ability with new unseen data. Smooth decision boundaries tended to generalize well, while convoluted and irregular decision boundaries did not.

Differences in the training time, classification time, and memory usage of the classifiers were often very large, even though their error rates did not differ significantly. For example, on the vowel problem, the Gaussian mixture and KD tree classifiers both provided low error rates, but the Gaussian mixture classifier required 20 times less classification memory than the KD tree classifier. In addition, the Gaussian mixture classifier took 10 times longer to train than the KD tree classifier.

There were trade-offs among the classifiers between different performance issues such as training time, classification time, and memory usage. In general, classifiers that had low classification memory requirements tended to require long training times, while those that trained rapidly were not memory efficient. In addition, classifiers that required a large amount of classification memory tended to require more time to perform classification.
Chapter 6

Summary and Future Directions

This study explored the behavior and practical characteristics of neural network and conventional pattern classifiers. Two neural network classifiers were studied (radial basis function and high order polynomial (GMDH) network classifiers). Four conventional classifiers were studied (linear decision tree, Gaussian mixture, KD tree, and condensed $K$ nearest neighbor classifiers). These classifiers were chosen to extend and complement those that were investigated in a previous study [12].

The results of these two studies demonstrate that classification error rates can be equivalent across different classifiers when they are powerful enough to form minimum error decision regions, when they are properly tuned, and when sufficient training data is available. Practical characteristics such as training time, classification time, and memory requirements, however, differed by orders of magnitude. In practice, these practical characteristics are likely to affect classifier selection and use substantially. Selection of a classifier for a particular problem or application will often be based more on practical considerations concerning memory usage, computational resources, and restrictions on training and classification times than on small differences in error rate.

The many existing neural network and conventional pattern classifiers trade these characteristics off. These trade-offs, however, depend on the particular implementation of the classifier and on the particular characteristics of the problem. Thus, it is important to be able to understand how classifiers behave, what factors affect classifier performance and behavior, and what practical characteristics different classifiers trade-off, so that guidelines can be developed for their proper selection and use.

In this study, it was found that the input dimensionality, the amount of training data, the number of classes, and the complexity of the decision regions required affected the performance of the classifiers in different ways. Radial basis function
classifiers generalized well in high dimensional spaces and provided low error rates with training times that were much less than those of back-propagation classifiers [12]. In addition, RBF classifiers formed smooth, well-behaved decision regions and were able to perform well with little training data. High order polynomial network classifiers provided intermediate error rates, but often required very long training times and large amounts of training memory. In addition, the decision regions formed did not extrapolate well to data poor regions of the input space. Gaussian mixture classifiers trained rapidly and provided good performance when the numbers and types of mixtures were selected carefully to model class densities. Linear tree classifiers were the most computationally efficient, but performed poorly for high dimensionality problems with small training set sizes. In addition, because of the overly complex tree that must be constructed during training, the linear tree classifier required large amounts of training memory. KD-tree classifiers reduced classification time by a factor of four over conventional K nearest neighbor (KNN) classifiers for low two-input dimension problems, but provided little or no reduction in classification time for high 22-input dimension problems. Condensed KNN classifiers reduced memory requirements over conventional KNN classifiers by a factor of two to fifteen for all problems and reduced classification times, without increasing the error rate significantly.

By understanding how classifiers behave and their limitations, a proper range of expectation can be set for different classifiers. Results from this work point in a direction of research focusing on constructing a broad knowledge base of understanding which classifiers perform well under what conditions. With this knowledge, a set of heuristics can then be formulated for selecting and using classifiers given a particular problem or set of constraints.

This study is just a small step towards understanding the behavior and trade-offs of practical characteristics of pattern classifiers. Further studies of neural network and conventional pattern classifiers are necessary to examine the behavior and practical characteristics in more complex, real world problems with more input dimensions, and more training data. In addition, more extensive studies should be performed to determine how to fine tune classifiers more rigorously. Although some classifiers in
this study were tuned formally using cross-validation and statistical methods, most of the classifiers were tuned through trial and error using both training and test data to obtain good performance. Better methods should be developed to fine-tune classifiers without using test data at all. Another avenue of investigation is to examine the issue of selecting and creating features for use in pattern recognition. Given an appropriate set of features for a problem, any pattern classifier can perform the required classification.
Bibliography


