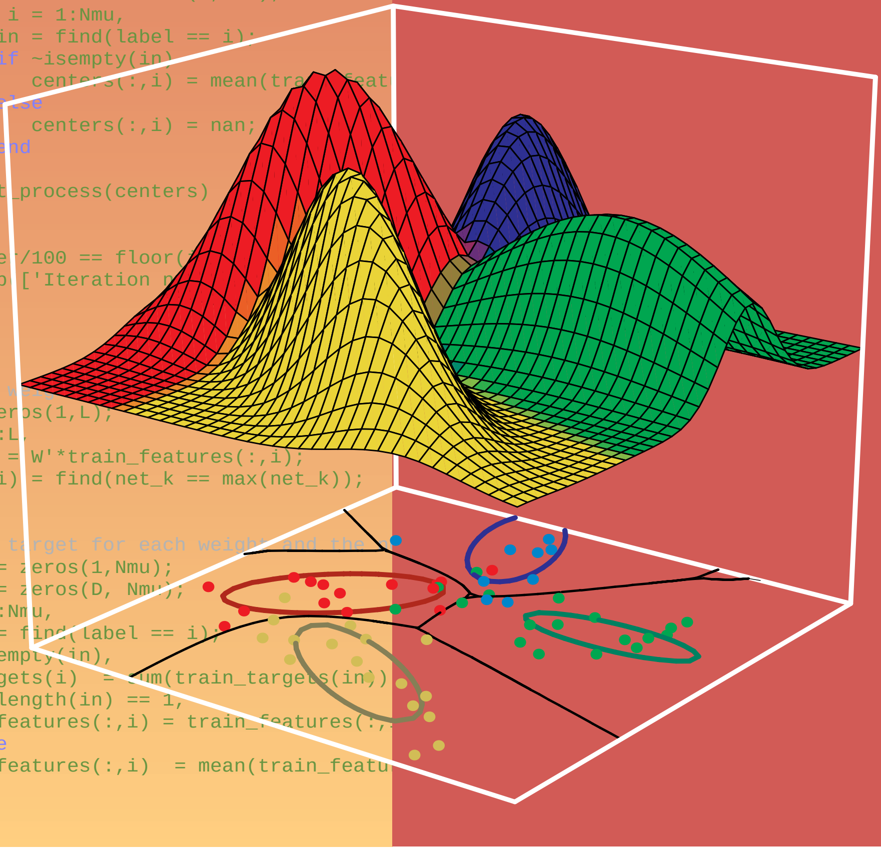
Computer Manual in MATLAB to accompany

Pattern Classification

David G. Stork
Elad Yom-Tov

```
 while(dW > 1e-15),
   %Choose a sample randomly
   i  = randperm(L);
   phi = train_features(:,i(1));
   net_k = W'*phi;
   y_star= find(net_k == max(net_k));
   y_star= y_star(1);

   %Just in case two have the same weights
   oldW = W;
   W  = W + eta*phi*gamma(win_width*abs(ne
   W  = W ./ (ones(D,1)*sqrt(sum(W.^2)));
   eta = eta * deta;
   dW  = sum(sum(abs(oldW-W)));
   iter = iter + 1;

if (plot_on == 1),
      %Assign each of the features to a ce
      dist         = W'*train_features;
      [m, label]  = max(dist);
      centers      = zeros(D,Nmu);
      for i = 1:Nmu,
         in = find(label == i);
         if ~isempty(in)
            centers(:,i) = mean(tra  feat
         else
            centers(:,i) = nan;
         end
      end
      plot_process(centers)
   end

   if (iter/100 == floor(
      disp(['Iteration n
   end

end

%Assign a weig
label = zeros(1,L);
for i = 1:L,
   net_k  = W'*train_features(:,i);
   label(i) = find(net_k == max(net_k));
end

%Find the target for each weight and the n
targets  = zeros(1,Nmu);
features = zeros(D, Nmu);
for i = 1:Nmu,
   in    = find(label == i);
   if ~isempty(in),
      targets(i)  = sum(train_targets(in))
      if length(in) == 1,
         features(:,i) = train_features(:,
      else
         features(:,i)  = mean(train_featu
      end
   end
end
```



Appendix to the Computer Manual in MATLAB to accompany Pattern Classification (2nd ed.)

David G. Stork and Elad Yom-Tov

By using the Classification toolbox you agree to the following licensing terms:

NO WARRANTY

THERE IS NO WARRANTY FOR THE PROGRAMS, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN THE WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAMS "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAMS ARE WITH YOU. SHOULD THE PROGRAMS PROVE DEFECTIVE, YOU ASSUME THECOST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAMS, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THEUSE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Contents

Preface

This Appendix is a pre-publication version to be included in the furthcoming version of the **Computer Manual to accompany Pattern Classification, 2nd Edition**. It includes short descriptions of the programs in the classification toolbox invoked directly by users.

Additional information and updates are available from th e authors' web site at `http://www.yom-tov.info`

We wish you the best of luck in your studies and research!

David G. Stork
Elad Yom-Tov

APPENDIX

Program descriptions

Below are short descriptions of the programs in the classification toolbox invoked directly by users. This listings are organized by chapter in **Pattern Classification**, and in some cases include pseudo-code. Not all programs here appear in the textbook and not every minor variant on an algorithm in the textbook appears here. While most classification programs take input data sets and targets, some classification and feature selection programs have associated additional inputs and outputs, as listed. You can obtain further specific information on the algorithms by consulting **Pattern Classification** and information on the MATLAB code by using its `help` command.

Chapter 2

Marginalization

Function name: `Marginalization`

Description:

Compute the marginal distribution of a multi-dimensional histogram or distribution as well as the marginal probabilities for test patterns given the “good” features.

Syntax:

```
predicted_targets = marginalization(training_patterns, training_targets, test_patterns, parameter vector);
```

Parameters:

1. The index of the missing feature.

2. The number of patterns with which to compute the marginal.

Minimum cost classifier

Function name: `minimum_cost`

Description:

Perform minimum-cost classification for known distributions and cost matrix λ_{ij}.

Syntax:

```
predicted_targets = minimum_cost(training_patterns, training_targets, test_patterns, parameter vector);
```

Parameter:

The cost matrix λ_{ij}.

Normal Density Discriminant Function

Function name: NNDF

Description:

Construct the Bayes classifier by computing the mean and d-by-d covariance matrix of each class and then use them to construct the Bayes decision region.

Syntax:

```
predicted_targets = NNDF(training_patterns, training_targets, test_patterns, parameter vector);
```

Parameters:

The discriminant function (probability) for any test pattern.

Stumps

Function name: `Stumps`

Description:

Determine the threshold value on a single feature that will yield the lowest training error. This classifier can be thought of as a linear classifier with a single weight that differs from zero.

Syntax:

```
predicted_targets = Stumps(training_patterns, training_targets, test_patterns, parameter vector);

[predicted_targets, weights] = Stumps(training_patterns, training_targets, test_patterns, parameter vector);
```

Parameter:

Optional: A weight vector for the training patterns.

Additional outputs:

The weight vector for the linear classifier arising from the optimal threshold value.

Discrete Bayes Classifier

Function name: `Discrete_Bayes`

Description:

Perform Bayesian classification on feature vectors having discrete values. In this implementation, discrete features are those that have no more than one decimal place. The program bins the data and then computes the probability of each class. The program then computes the classification decision based on standard Bayes theory.

Syntax:

predicted_targets = Discrete_Bayes(training_patterns, training_targets, test_patterns, parameter vector);

Parameters:

None

Multiple Discriminant Analysis

Function name: `MultipleDiscriminantAnalysis`

Description:

Find the discriminants for a multi-category problem. The discriminant maximizes the ratio of the between-class variance to that of the in-class variance.

Syntax:

```
[new_patterns, new_targets] = MultipleDiscriminantAnalysis(training_patterns, training_targets);

[new_patterns, new_targets, feature_weights] = MultipleDiscriminantAnalysis(training_patterns, training_targets);
```

Additional outputs:

The weight vectors for the discriminant boundaries.

Bhattacharyya

Function name: `Bhattacharyya`

Description:

Estimate the Bhattacharyya error rate for a two-category problem, assuing Gaussianity. The bound is given by:

$$k\left(\frac{1}{2}\right) = \frac{1}{8}(\mu_1 - \mu_2)^t(\Sigma_1 - \Sigma_2)^{-1}(\mu_1 - \mu_2) + \frac{1}{2}\ln\frac{|\Sigma_1 - \Sigma_2|}{2\sqrt{|\Sigma_1||\Sigma_2|}}$$

Syntax:

error_bound = Bhattacharyya(mu1, sigma1, mu2, sigma2, p1);

Input variables:

1. mu1, mu2 - The means of class 1 and 2, respectively.

2. sigma1, sigma2 - The covariance of class 1 and 2, respectively.

3. p1 - The probability of class 1.

Chernoff

Function name: `Chernoff`

Description:

Estimate the Chernoff error rate for a two-category problem. The error rate is computed through the following equation:

$$min_{\beta}\left\{e^{-\left[\beta\frac{(1-\beta)}{2}(\mu_2-\mu_1)^T[\beta\Sigma_1+(1-\beta)\Sigma_2]^{-1}(\mu_2-\mu_1)+\frac{1}{2}\ln\left|\frac{\beta\Sigma_1+(1-\beta)\Sigma_2}{|\Sigma_1|^{\beta}|\Sigma_2|^{1-\beta}}\right|\right]}\right\}$$

Syntax:

```
error_bound = Chernoff(mu1, sigma1, mu2, sigma2, p1);
```

Input variables:

1. mu1, mu2 - The means of class 1 and 2, respectively.

2. sigma1, sigma2 - The covariance of class 1 and 2, respectively.

3. p1 - The probability of class 1.

Discriminability

Funciton name: `Discriminability`

Description: Compute the discriminability d' in the Receiver Operating Characteristic (ROC) curve.

Syntax:

```
d_tag = Discriminability(mu1, sigma1, mu2, sigma2, p1);
```

Input variables:

1. mu1, mu2 - The means of class 1 and 2, respectively.

2. sigma1, sigma2 - The covariance of class 1 and 2, respectively.

3. p1 - The probability of class 1.

Chapter 3

Maximum-Likelihood Classifier

Function name: `ML`

Description:

Compute the maximum-likelihood estimate of the mean and covariance matrix of each class and then uses the results to construct the Bayes decision region. This classifier works well if the classes are uni-modal, even when they are not linearly seperable.

Syntax:

```
predicted_targets = ML(training_patterns, training_targets, test_patterns, []);
```

Maximum-Likelihood Classifier assuming Diagonal Covariance Matrices

Function name: `ML_diag`

Description:

Compute the maximum-likelihood estimate of the mean and covariance matrix (assumed diagonal) of each class and then uses the results to construct the Bayes decision region. This classifier works well if the classes are unimodal, even when they are not linearly seperable.

Syntax:

```
predicted_targets = ML_diag(training_patterns, training_targets, test_patterns,  []);
```

Gibbs

Function name: `Gibbs`

Description:

This program finds the probability that the training data comes from a Gaussian distribution with known parameters, i.e., $P(D|\theta)$. Then, using $P(D|\theta)$, the program samples the parameters according to the Gibbs method, and finally uses the parameters to classify the test patterns.

Syntax:

```
predicted_targets = Discrete_Bayes(training_patterns, training_targets, test_patterns, input parameter);
```

Parameter:

Resolution of the input features (i.e., the number of bins).

Fishers Linear Discriminant

Function name: `FishersLinearDiscriminant`

Description:

Computes the Fisher linear discriminant for a pair of distributions. The Fisher linear discriminant attempts to maximize the ratio of the between-class variance to that of the in-class variance. This is done by reshaping the data through a linear weight vector computed by the equasion:

$$w = S_W^{-1}(m_1 - m_2)$$

where S_W is the in-class (or within-class) scatter matrix.

Syntax:

```
[new_patterns, new_targets] = FishersLinearDiscriminant(training_patterns, training_targets, [], []);

[new_patterns, new_targets, weights] = FishersLinearDiscriminant(training_patterns, training_targets, [], []);
```

Additional outputs:

The weight vector for the linear classifier.

Local Polynomial Classifier

Function name: `Local_Polynomial`

Description: This nonlinear classification algorithm works by building a classifier based on a local subset of training points, and classifies the test points according to those local classifiers. The method randomly selects a predetermined number of the training points and then assign each of the test points to the nearest of the points so selected. Next, the method builds a logistic classifier around these selected points, and finally classifies the points assigned to it.

Syntax:

```
predicted_targets = Local_Polynomial(training_patterns, training_targets, test_patterns, input parameter);
```

Input parameter:

Number of (local) points to select for creation of a local polynomial or logistic classifier.

Expectation-Maximization

Function name: `Expectation_Maximization`

Description:

Estimate the means and covariances of component Gaussians by the method of expectation-maximization.

Pseudo-code:

```
begin initialize $\theta_0, T, i \leftarrow 0$
        do $i \leftarrow i + 1$
            E step: compute $Q(\theta;\theta^i)$
            M step: $\theta^{i+1} \leftarrow argmax_{\theta} Q(\theta;\theta^i)$
        until $Q(\theta^{i+1};\theta^i) - Q(\theta^i;\theta^{i-1}) \le T$
    return $\hat{\theta} \leftarrow \theta^{i+1}$
end
```

Syntax:

predicted_targets = EM(training_patterns, training_targets, test_patterns, input parameters);

[predicted_targets, estimated_parameters] = EM(training_patterns, training_targets, test_patterns, input parameters);

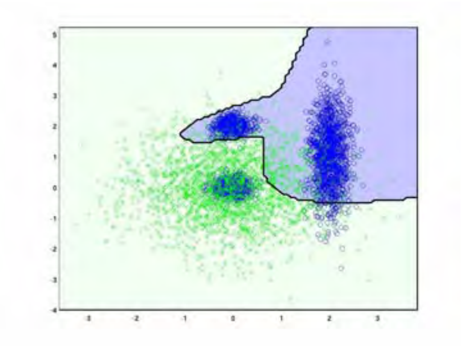
Input parameters:

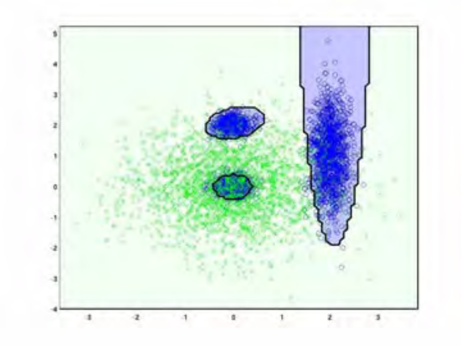
The number of Gaussians for each class.

Additional outputs:

The estimated means and covariances of Gaussians.

Example:





These figures show the results of running the EM algorithm with different parameter values. The left figure shows the decision region obtained when the wrong number of Gaussians is entered, while the right shows the decision region when the correct number of Gaussians in each class is entered.

Multivariate Spline Classification

Function name: `Multivariate_Splines`

Description:

This algorithm fits a spline to the histogram of each of the features of the data. The algorithm then selects the spline that reduces the training error the most, and computes the associated residual of the prediction error. The process iterates on the remaining features, until all have been used. Then, the prediction of each spline is evaluated independently, and the weight of each spline is computed via the pseudo-inverse. This algorithm is typically used for regression but here is used for classification.

Syntax:

```
predicted_targets = Multivariate_Splines(training_patterns, training_targets, test_patterns, input parameters);
```

Input parameters:

1. The degree of the splines.

2. The number of knots per spline.

Whitening transform

Function name: `Whitening_transform`

Description:

Apply the whitening transform to a d-dimensional data set. The algorithm first subtracts the sample mean from each point, and then multiplies the data set by the inverse of the square root of the covariance matrix.

Syntax:

```
[new_patterns, new_targets] = Whitening_transform(training_patterns, training_targets, [], []);

[new_patterns, new_targets, means, whiten_mat] = Whitening_transform(training_patterns, training_targets, [], []);
```

Additional outputs:

1. The whitening matrix.

2. The means vector.

Scaling transform

Function name: `Scaling_transform`

Description:

Standardize the data, that is, transforms a data set so that it has zero mean and unit variance along each coordinate. This scaling is recommended as preprocessing for data presented to a neural network classifier.

Syntax:

```
[new_patterns, new_targets] = Scaling_transform(training_patterns, training_targets, [], []);

[new_patterns, new_targets, means, variance_mat] = Scaling_transform(training_patterns, training_targets, [], []);
```

Additional outputs:

1. The variance matrix.

2. The means vector.

Hidden Markov Model Forward Algorithm

Function name: `HMM_Forward`

Description:

Compute the probability that a test sequence $\boldsymbol{V}^T$ was generated by a given hidden Markov model according to the Forward algorithm. Note: This algorithm is in the “Other” subdirectory.

Pseudo-code:

```
begin initialize $t \leftarrow 0$, $a_{ij}$, $b_{jk}$, visible sequence $\mathbf{V}^{\mathrm{T}}$, $\alpha_j(0)$
            for  $t \leftarrow t + 1$
                  $\beta_i(t) \leftarrow \sum_{i=1}^{c} \beta_j(t+1) a_{ij} b_{jk} v(t+1)$
            until t=T
      return $P(\boldsymbol{V}^T) \leftarrow \alpha_0(T)$   for the final state
end
```

Syntax:

[Probability_matrix, Probability_matrix_through_estimation_stages] =
HMM_Forward(Transition_prob_matrix, Output_generation_mat, Initial_state, Observed output sequence);

Hidden Markov Model Backward Algorithm

Function name: `HMM_Backward`

Description:

Compute the probability that a test sequence $\boldsymbol{V}^T$ was generated by a given hidden Markov model according to the Backward algorithm. Learning in hidden Markov models via the Forward-Backward algorithm makes use of both the Forward and the Backward algorithms. Note: This algorithm is in the "Other" subdirectory.

Pseudo-code:

```
begin initialize $\beta_j(\mathrm{T})$, $t \leftarrow T$, $a_{ij}$, $b_{jk}$, visible sequence $\mathbf{V}^{\mathrm{T}}$
        for  $t \leftarrow t - 1$
                $\beta_i(t) \leftarrow \sum_{i=1}^{c} \beta_j(t+1) a_{ij} b_{jk} v(t+1)$
        until t=1
    return $P(\boldsymbol{V}^T) \leftarrow \beta_i(0)$    for the known initial state
end
```

Syntax:

[Probability_matrix, Probability_matrix_through_estimation_stages] =
HMM_Backward(Transition_prob_matrix, Output_generation_mat, Final_state, Observed output sequence);

Forward-Backward Algorithm

Function name: HMM_Forward_Backward

Description:

Estimate the parameters in a hidden Markov model based on a set of training sequences. Note: This algorithm is in the “Other” subdirectory.

Pseudo-code:

```
begin initialize $a_{ij}$, $b_{jk}$, training sequence $\mathbf{V}^T$, convergence criterion $\theta$, $z \leftarrow 0$
        do $z \leftarrow z + 1$
            compute $\hat{a}(z)$ from $a(z-1)$ and $b(z-1)$
            compute $\hat{b}(z)$ from $a(z-1)$ and $b(z-1)$
            $a_{ij}(z) \leftarrow \hat{a}_{ij}(z-1)$
            $b_{jk}(z) \leftarrow \hat{b}_{jk}(z-1)$
        until $max_{i,j,k}\left[a_{ij}(z) - a_{ij}(z-1), b_{jk}(z) - b_{jk}(z-1)\right] < \theta$
    return $a_{ij} \leftarrow a_{ij}(z)$, $b_{jk} \leftarrow b_{jk}(z)$
end
```

Syntax:

[Estimated_Transition_Probability_matrix, Estimated_Output_Generation_matrix] =
HMM_Forward_backward(Transition_prob_matrix, Output_generation_mat, Observed output sequence);

Hidden Markov Model Decoding

Function name: `HMM_Decoding`

Description:

Estimate a highly likely path through the hidden Markov model (trellis) based on the topology and transition probabilities in that model. Note: This algorithm is in the “Other” subdirectory.

Pseudo-code:

```
begin initialize $Path \leftarrow \{\ldots\}$, $t \leftarrow 0$
    for $t \leftarrow t + 1$
        for $j \leftarrow j + 1$
            $\alpha_j(t) \leftarrow b_{jk} v(t) \sum_{i=1}^{c} \alpha_i(t-1) a_{ij}$
        until $j = c$
        $j' \leftarrow \arg max_j \alpha_j(t)$
        Append $\omega_{j'}$ to Path
        until $t = T$
    return Path
end
```

Syntax:

Likely_sequence = HMM_Forward(Transition_prob_matrix, Output_generation_mat, Initial_state, Observed output sequence);

Chapter 4

Nearest-Neighbor Classifier

Function name: `Nearest_Neighbor`

Description:

For each of the test examples, the nearest k neighbors from training examples are found, and the majority label among these are given as the label to the test example. The number of nearest neighbors determines how local the classifier is. If this number is small, the classifier is more localized. This classifier usually results in reasonably low training error, but it is expensive computationally and memory-wise.

Syntax:

```
predicted_targets = Nearest_Neighbor(training_patterns, training_targets, test_patterns, input parameter);
```

Input parameters:

Number of nearest neighbors, k.

Nearest-Neighbor Editing

Function name: `NearestNeighborEditing`

Description:

This algorithm searches for the Voronoi neighbors of each pattern. If the labels of all the neighbors are the same, the pattern in discarded. The MATLAB implementation uses linear programming to increase speed. This algorithm can be used for reducing the number of training data points.

Pseudo-code:

```
begin initialize $j \leftarrow 0$, $D \leftarrow data\ set$, $n \leftarrow num\ prototypes$
            construct the full Voronoid diagram of D
            do $j \leftarrow j + 1$ ; for each prototype $x_j'$
                  find the Voronoi neighbors of $x_j'$
             if any neighbor is not from the same class as $x_j'$ then mark $x_j'$
      until $j = n$
      discard all points that are not marked
      construct the Voronoi diagram of the remaining (marked) prototypes
end
```

Syntax:

```
[new_patterns, new_targets] = NearestNeighborEditing(training_patterns, training_targets, [], []);
```

Store-Grabbag Algorithm

Function name: `Store_Grabbag`

Description:

The store-grabbag algorithm is a modification of the nearest-neighbor algorithm. The algorithm identifies those samples in the training set that affect the classification, and discards the others.

Syntax:

```
predicted_targets = Store_Grabbag(training_patterns, training_targets, test_patterns, input parameter);
```

Input parameter:

Number of nearest neighbors, k.

Reduced Coloumb Energy

Function name: `RCE`

Description: Create a classifier based on a training set, maximizing the radius around each training point (up to λ_{max}) yet not misclassifying other training points.

Pseudo-code:

Training

```
begin initialize $j \leftarrow 0$, $n \leftarrow num\ patterns$, $\varepsilon \leftarrow small\,param$, $\lambda_m \leftarrow max\ radius$
          do   $j \leftarrow j + 1$
                    $w_{ij} \leftarrow x_i$ (train weight)
                    $\hat{\mathbf{x}} \leftarrow \arg min_{\mathbf{x} \notin \omega_i} D(\mathbf{x}, \mathbf{x}')$ (find nearest point not in $\omega_i$)
                    $\lambda_j \leftarrow min\left[D(\hat{\mathbf{x}}, \mathbf{x}') - \varepsilon, \lambda_m\right]$ (set radius)
          if $x \in \omega_k$ then $a_{jk} \leftarrow 1$
     until $j = n$
end
```

Classification

```
begin initialize $j \leftarrow 0$ , $k \leftarrow 0$ , $\boldsymbol{x} \leftarrow test\ pattern$, $\mathrm{D}_t \leftarrow \{\ldots\}$
          do $j \leftarrow j + 1$
               if $D(\boldsymbol{x}, \boldsymbol{x}_j') < \lambda_j$ then $\mathrm{D}_t \leftarrow \mathrm{D}_t \cup \boldsymbol{x}'_j$
          until $j = n$
          if label of all $\boldsymbol{x}'_j \in \mathrm{D}_t$ is the same then return label of all $\boldsymbol{x}_k \in \mathrm{D}_t$
                         else return “ambiguous” label
end
```

Syntax:

predicted_targets = RCE(training_patterns, training_targets, test_patterns, input parameter);

Input parameters:

The maximum allowable radius, λ_{max}.

Parzen Windows Classifier

Function name: `Parzen`

Description:

Estimate a posterior density by convolving the data set in each category with a Gaussian Parzen window of scale h. The scale of the window determines the locality of the classifier such that a larger h causes the classifier to be more global.

Syntax:

```
predicted_targets = Parzen(training_patterns, training_targets, test_patterns, input parameter);
```

Input parameter:

Normalizing factor for the window width, h.

Probabilistic Neural Network Classification

Function name: PNN

Description:

This algorithm trains a probabalistic neural network and uses it to classify test data. The PNN is a parallel implementation of the Parzen windows classifier.

Pseudo-code

```
begin initialize $k \leftarrow 0$, $\mathbf{x} \leftarrow test\ pattern$
          do $k \leftarrow k + 1$
               $net_k \leftarrow \mathbf{w}_k^t \mathbf{x}$
               if $a_{ki} = 1$ then $g_i \leftarrow g_i + \exp[(net_k - 1)/\sigma^2]$
     return $class \leftarrow argmax_i\ g_i(\mathbf{x})$
end
```

Syntax:

predicted_targets = PNN(training_patterns, training_targets, test_patterns, input parameter);

Input parameter:

The Gaussian width, σ.

Chapter 5

Basic Gradient Descent

Function name: `BasicGradientDescent`

Description:

Perform simple gradient descent in a scalar-valued criterion function $J(\boldsymbol{a})$.

Pseudo-code:

```
begin initialize $\boldsymbol{a}$, threshold $\theta$, $\eta(.)$, $k \leftarrow 0$
        do $k \leftarrow k + 1$
            $\boldsymbol{a} \leftarrow \boldsymbol{a} - \eta(k)\nabla J(\boldsymbol{a})$
        until $|\eta(k)\nabla J(\boldsymbol{a})| < \theta$
    return $\boldsymbol{a}$
end
```

Syntax:

min_point = gradient_descent(Initial search point, theta, eta, function to minimize)

Note: The function to minimize must accept a value and return the function's value at that point.

Newton Gradient Descent

Function name: `Newton_descent`

Description:

Perform Newton's method for gradient descent in a scalar-valued criterion function $J(\boldsymbol{a})$, where the Hessian matrix $\boldsymbol{H}$ can be computed.

Pseudo-code:

```
begin initialize $\boldsymbol{a}$, threshold $\theta$
        do
              $\boldsymbol{a} \leftarrow \boldsymbol{a} - \boldsymbol{H}^{-1}\nabla J(\boldsymbol{a})$
        until $|\boldsymbol{H}^{-1}\nabla J(\boldsymbol{a})| < \theta$
    return $\boldsymbol{a}$
end
```

Syntax:

min_point = Newton_descent(Initial search point, theta, function to minimize)

Note: The function to minimize must accept a value and return the function's value at that point.

Batch Perceptron

Function name: `Perceptron_Batch`

Description:

Train a linear Perceptron classifier in batch mode.

Pseudo-code:

```
begin initialize $\boldsymbol{a}$, criterion $\theta$, $\eta(.)$, $k \leftarrow 0$
            do $k \leftarrow k + 1$
                  $\boldsymbol{a} \leftarrow \boldsymbol{a} + \eta(k) \sum_{y \in \mathrm{Y}_k} \boldsymbol{y}$
            until $\left| \eta(k) \sum_{y \in \mathrm{Y}} \boldsymbol{y} \right| < \theta$
      return $\boldsymbol{a}$
end
```

Syntax:

predicted_targets = Perceptron_Batch(training_patterns, training_targets, test_patterns, input parameters);

[predicted_targets, weights] = Perceptron_Batch(training_patterns, training_targets, test_patterns, input parameters);

[predicted_targets, weights, weights_through_the_training] = Perceptron_Batch(training_patterns, training_targets, test_patterns, input parameters);

Input parameters:

1. The maximum number of iterations.

2. The convergence criterion.

3. The convergence rate.

Additional outputs:

1. The weight vector for the linear classifier.

2. The weights throughout learning.

Fixed-Increment Single-Sample Perceptron

Function name: `Perceptron_FIS`

Description:

This algorithm attempts to iteratively find a linear separating hyperplane. If the problem is linear, the algorithm is guaranteed to find a solution. During the iterative learning process the algorithm randomly selects a sample from the training set and tests if that sample is correctly classified. If not, the weight vector of the classifier is updated. The algorithm iterates until all training samples are correctly classified or the maximal number of training iterations is reached.

Pseudo-code:

```
begin initialize $\boldsymbol{a}, k \leftarrow 0$
      do $k \leftarrow (k+1) mod\ n$
            if $\mathbf{y}^k$ is misclassified by a then $\boldsymbol{a} \leftarrow \boldsymbol{a} + \boldsymbol{y}^k$
            until all patterns properly classified
      return a
end
```

Syntax:

predicted_targets = Perceptron_FIS(training_patterns, training_targets, test_patterns, input parameter);

[predicted_targets, weights] = Perceptron_FIS(training_patterns, training_targets, test_patterns, input parameter);

Input parameters:

The parameters describing either the maximum number of iterations, or a weight vector for the training samples, or both.

Additional outputs:

The weight vector for the linear classifier.

Variable-increment Perceptron with Margin

Function name: `Perceptron_VIM`

Description:

This algorithm trains a linear Perceptron classifier with a margin by adjusting the weight step size.

Pseudo-code

```
begin initialize $\boldsymbol{a}$, threshold $\theta$, margin $b$, $\eta(.)$, $k \leftarrow 0$
    do $k \leftarrow (k+1) mod\ n$
        if $\boldsymbol{a}^t\boldsymbol{y}^k \le b$ then $\boldsymbol{a} \leftarrow \boldsymbol{a} + \eta(k)\boldsymbol{y}^k$
        until $\boldsymbol{a}^t\boldsymbol{y}^k > b$ for all $k$
    return a
end
```

Syntax:

predicted_targets = Perceptron_VIM(training_patterns, training_targets, test_patterns, input parameter);

[predicted_targets, weights] = Perceptron_VIM(training_patterns, training_targets, test_patterns, input parameter);

Additional inputs:

1. The margin *b*.

2. The maximum number of iterations.

3. The convergence criterion.

4. The convergence rate.

Additional outputs:

The weight vector for the linear classifier.

Batch Variable Increment Perceptron

Function name: `Perceptron_BVI`

Description:

This algorithm trains a linear Perceptron classifier in the batch mode, and where the learning rate is variable.

Pseudo-code:

```
begin initialize $\boldsymbol{a}, \eta(.), k \leftarrow 0$
    do $k \leftarrow (k+1) mod\ n$
        $Y_k = \{\}$
        $j = 0$
        do $j \leftarrow j + 1$
            if $\mathbf{y}_j$ is misclassified then Append $\mathbf{y}_j$ is to $Y_k$
        until $j = n$
        $\boldsymbol{a} \leftarrow \boldsymbol{a} + \eta(k) \sum_{\boldsymbol{y} \in Y_k} \boldsymbol{y}$
        until $Y_k = \{\}$
    return a
end
```

Syntax:

```
predicted_targets = Perceptron_BVI(training_patterns, training_targets, test_patterns, input parameter);

[predicted_targets, weights] = Perceptron_BVI(training_patterns, training_targets, test_patterns, input parameter);
```

Input parameters:

Either the maximum number of iterations, or a weight vector for the training samples, or both.

Additional outputs:

The weight vector for the linear classifier.

Balanced Winnow

Function name: `Balanced_Winnow`

Description:

This algorithm implements the balanced Winnow algorithm, which uses both a positive and negative weight vectors, each adjusted toward the final decision boundary from opposite sides.

Pseudo-code:

```
begin initialize $\mathbf{a}^+$, $\mathbf{a}^-$, $\eta(.)$, $k \leftarrow 0$, $\alpha > 1$
    if Sgn[$\mathbf{a}^{+t}\mathbf{y}_k$ - $\mathbf{a}^{-t}\mathbf{y}_k$ ] ¦ $z_k$ (pattern misclassified)
    then if $z_k = +1$ then $a_i^\dagger \leftarrow \alpha^{y_i} a_i^\dagger$ ; $a_i^{-\circ} \leftarrow \alpha^{-y_i} a_i^{-\circ}$   for all $i$
         if $z_k = -1$ then $a_i^\dagger \leftarrow \alpha^{-y_i} a_i^\dagger$;  $a_i^{-\circ} \leftarrow \alpha^{y_i} a_i^{-\circ}$   for all $i$
    return $\mathbf{a}^+$, $\mathbf{a}^-$
end
```

Syntax:

predicted_targets = Balanced_Winnow(training_patterns, training_targets, test_patterns, input parameters);

[predicted_targets, positive_weights, negative_weights] = Balanced_Winnow(training_patterns, training_targets, test_patterns, input parameters);

Input parameters:

1. The maximum number of iterations.

2. The scaling parameter, alpha.

3. The convergence rate, eta.

Additional outputs:

The positive weight vector and the negative weight vector.

Batch Relaxation with Margin

Function name: `Relaxation_BM`

Description: This algorithm trains a linear Perceptron classifier with margin *b* in the batch mode.

Pseudo-code:

```
begin initialize $\boldsymbol{a}, \eta(.), b, k \leftarrow 0$
    do $k \leftarrow (k+1) mod\ n$
        $Y_k = \{\}$
        $j = 0$
        do $j \leftarrow j + 1$
            if $\boldsymbol{a}^t y^j \leq b$ then Append $\mathbf{y}_j$ is to $Y_k$
        until $j = n$
        $\boldsymbol{a} \leftarrow \boldsymbol{a} + \eta(k) \sum_{\boldsymbol{y} \in Y_k} \frac{b - \boldsymbol{a}^t \boldsymbol{y}}{|\boldsymbol{y}|^2}$
        until $Y_k = \{\}$
    return a
end
```

Syntax:

```
predicted_targets = Relaxation_BM(training_patterns, training_targets, test_patterns, input parameters);

[predicted_targets, weights] = Relaxation_BM(training_patterns, training_targets, test_patterns, input parameters);
```

Input parameters:

1. The maximum number of iterations.

2. The target margin, *b*.

3. The convergence rate, eta.

Additional outputs:

The weight vector for the final linear classifier.

Single-Sample Relaxation with Margin

Function name: `Relaxation_SSM`

Description:

This algorithm trains a linear Perceptron classifier with margin on a per-pattern basis.

Pseudo-code

```
begin initialize $\boldsymbol{a}, b, \eta(.), k \leftarrow 0$
    do $k \leftarrow (k+1) mod\ n$
            if $\boldsymbol{a}^t y^j \leq b$ then $\boldsymbol{a} \leftarrow \boldsymbol{a} + \eta(k)\frac{b - \boldsymbol{a}^t\boldsymbol{y}^k}{|\boldsymbol{y}^k|^2}\boldsymbol{y}^k$
        until $\mathbf{a}^t\mathbf{y}^k > b$ for all $\mathbf{y}^k$
    return a
end
```

Syntax:

predicted_targets = Relaxation_SSM(training_patterns, training_targets, test_patterns, input parameters);

[predicted_targets, weights] = Relaxation_SSM(training_patterns, training_targets, test_patterns, input parameters);

Input parameters:

1. The maximum number of iterations.

2. The margin, b.

3. The convergence rate, eta.

Additional outputs:

The weight vector for the final linear classifier.

Least-Mean Square

Function name: `LMS`

Description:

This algorithm trains a linear Perceptron classifier using the least-mean square algorithm.

Pseudo-code

```
begin initialize $\boldsymbol{a}$, $\boldsymbol{b}$, threshold $\theta$, $\eta(.)$, $k \leftarrow 0$
    do $k \leftarrow (k+1) mod\ n$
        $\boldsymbol{a} \leftarrow \boldsymbol{a} + \eta(k)(b_k - \boldsymbol{a}^t\boldsymbol{y}^k)\boldsymbol{y}^k$
        until $|\eta(k)(b_k - \boldsymbol{a}^t\boldsymbol{y}^k)| < \theta$
    return $\boldsymbol{a}$
end
```

Syntax:

predicted_targets = LMS(training_patterns, training_targets, test_patterns, input parameters);

[predicted_targets, weights] = LMS(training_patterns, training_targets, test_patterns, input parameters);

[predicted_targets, weights, weights_through_the_training] = LMS(training_patterns, training_targets, test_patterns, input parameters);

Input parameters:

1. The maximum number of iterations.

2. The convergence criterion.

3. The convergence rate.

Additional outputs:

1. The final weight vector.

2. The weight vector throughout the training procedure.

Least-Squares Classifier

Function name: LS

Description:

This algorithm trains a linear classifier by computing the weight vector using the Moore-Penrose pseudo-inverse, i.e.:

$$w = (PP^T)^{-1}PT^T$$

where *P* is the pattern matrix and *T* the target vector.

Syntax:

```
predicted_targets = LS(training_patterns, training_targets, test_patterns, input parameter);

[predicted_targets, weights] = LS(training_patterns, training_targets, test_patterns, input parameter);
```

Input parameters:

An optional weight vector for *weighted* least squares.

Additional outputs:

The weight vector of the final trained classifier.

Ho-Kashyap

Function name: `Ho_Kashyap`

Description:

This algorithm trains a linear classifier by the Ho-Kashyap algorithm.

Pseudo-code

Regular Ho-Kashyap

```
begin initialize $\boldsymbol{a}$, $\boldsymbol{b}$, $\eta(.) < 1$, threshold $b_{min}$, $k_{max}$
        do $k \leftarrow (k + 1) mod\ n$
                $\boldsymbol{e} \leftarrow \boldsymbol{Ya} - \boldsymbol{b}$
                $\boldsymbol{e}^{\dagger} \leftarrow 1/2(\boldsymbol{e} + Abs(\boldsymbol{e}))$
                $\boldsymbol{b} \leftarrow \boldsymbol{b} + 2\eta(k)\boldsymbol{e}^{\dagger}$
                $\boldsymbol{a} \leftarrow \boldsymbol{Y}^{\dagger}\boldsymbol{b}$
                if Abs(e) ð $b_{min}$ then return $\boldsymbol{a}$, $\boldsymbol{b}$ and exit
        until $k = k_{max}$
Print “NO SOLUTION FOUND”
end
```

Modified Ho-Kashyap

```
begin initialize $\boldsymbol{a}$, $\boldsymbol{b}$, $\eta < 1$, threshold $b_{min}$, $k_{max}$
```

```
    do $k \leftarrow (k+1) mod\ n$
        $\boldsymbol{e} \leftarrow \boldsymbol{Ya} - \boldsymbol{b}$
        $\boldsymbol{e}^{\dagger} \leftarrow 1/2(\boldsymbol{e} + Abs(\boldsymbol{e}))$
        $\boldsymbol{b} \leftarrow \boldsymbol{b} + 2\eta(k)(\boldsymbol{e} + Abs(\boldsymbol{e}))$
        $\boldsymbol{a} \leftarrow \boldsymbol{Y}^{\dagger}\boldsymbol{b}$
        if Abs(e) ð $b_{min}$ then return $\boldsymbol{a}$, $\boldsymbol{b}$ and exit
    until $k = k_{max}$
Print "NO SOLUTION FOUND"
end
```

Syntax:

predicted_targets = Ho_Kashyap(training_patterns, training_targets, test_patterns, input parameters);

[predicted_targets, weights] = Ho_Kashyap(training_patterns, training_targets, test_patterns, input parameters);

[predicted_targets, weights, final_margin] = Ho_Kashyap(training_patterns, training_targets, test_patterns, input parameters);

Additional inputs:

1. The type of training (`Basic` or `modified`).

2. The maximum number of iterations.

3. The convergence criterion.

4. The convergence rate.

Additional outputs:

1. The weights for the linear classifier.

2. The final computed margin.

Voted Perceptron Classifier

Function name: `Perceptron_Voted`

Description:

The voted Perceptron is a variant of the Perceptron where, in this implementation, the data may be transformed using a kernel function so as to increase the separation between classes.

Syntax:

```
predicted_targets = Perceptron_Voted(training_patterns, training_targets, test_patterns, input parameters);
```

Input parameters:

1. Number of perceptrons.

2. Kernel type: Linear, Polynomial, or Gaussian.

3. Kernel parameters.

Pocket Algorithm

Function name: `Pocket`

Description:

The pocket algorithm is a simple modification over the Perceptron algorithm. The improvement is that updates to the weight vector are retained only if they perform better on a random sample of the data. In the current MATLAB implementation, the weight vector is trained for 10 iterations. Then, the new weight vector and the previous weight vector are used to train randomly selected training patterns. If the new weight vector succeeded in classifying more patterns before it misclassified a pattern compared to the old weight vector, the new weight vector replaces the old weight vector. The procedure is repeated until convergence or the maximum number of iterations is reached.

Syntax:

predicted_targets = Pocket(training_patterns, training_targets, test_patterns, input parameters);

[predicted_targets, weights] = Pocket(training_patterns, training_targets, test_patterns, input parameters);

Input parameters:

Either the maximal number of iterations or weight vector for the training samples, or both.

Additional outputs:

The weight vector for the final linear classifier.

Farthest-margin perceptron

Function name: `Perceptron_FM`

Description:

This algorithm implements a slight variation on the traditional Perceptron algorithm, with the only difference that the wrongly classified sample *farthest* from the current decision boundary is used to adjust the weight of the classifier.

Syntax:

predicted_targets = Perceptron_FM(training_patterns, training_targets, test_patterns, input parameters);

[predicted_targets, weights] = Perceptron_FM(training_patterns, training_targets, test_patterns, input parameters);

Input parameters:

1. The maximum number of iterations.

2. The slack for incorrectly classified examples

Additional outputs:

The weight vector for the trained linear classifier.

Support Vector Machine

Function name: `SVM`

Description:

This algorithm implements a support vector machine and works in two stages. In the first stage, the algorithm transforms the data by a kernel function; in the second stage, the algorithm finds a linear separating hyperplane in kernel space. The first stage depends on the selected kernel function and the second stage depends on the algorithmic `solver` method selected by the user. The solver can be a quadratic programming algorithm, a simple farthest-margin Perceptron, or the Lagrangian algorithm. The number of support vectors found will usually be larger than is actually needed if the first two solvers are used because both solvers are approximate.

Syntax:

```
predicted_targets = SVM(training_patterns, training_targets, test_patterns, input parameters);

[predicted_targets, alphas] = SVM(training_patterns, training_targets, test_patterns, input parameters);
```

Input parameters:

1. The kernel function: `Gauss` (or `RBF`), `Poly`, `Sigmoid`, or `Linear`.
2. Kernel parameter: For each kernel parameters the following parameters are needed:
- RBF kernel: Gaussian width (scalar parameter)
- Poly kernel: The integer degree of the polynomial
- Sigmoid: The slope and constant of the sigmoid
- Linear: no parameters are needed
3. The choice of solver: `Perceptron`, `Quadprog`, or `Lagrangian`.
4. The slack, or tolerance.

Additional outputs:

The SVM coefficients.

Regularized Descriminant Analysis

Function name: RDA

Description:

This algorithm functions much as does the ML algorithm. However, once the mean and covariance of Gaussians are estimated they are shrunk.

Syntax:

```
predicted_targets = RDA(training_patterns, training_targets, test_patterns, input parameter);
```

Input parameter:

The shrinkage coefficient.

Reference:

J. Friedman, "Regularized discriminant analysis," *Journal of the American Statistical Association*, **84**:165-75 (1989)

Chapter 6

Stochastic Backpropagation

Function name: `Backpropagation_Stochastic`

Description:

This algorithm implements the stochastic backpropagation learning algorithm in a three-layer network of nonlinear units.

Pseudo-code:

```
begin initialize $n_H$, $\mathbf{w}$, criterion $\theta$, $\eta$, $m \leftarrow 0$
            do $m \leftarrow m + 1$
                $\mathbf{x}^m \leftarrow randomly\ chosen\ pattern$
                $w_{ji} \leftarrow w_{ji} + \eta\delta_j x_i$ ; $w_{kj} \leftarrow w_{kj} + \eta\delta_k y_j$
            until $\|\nabla J(\mathbf{w})\| < \theta$
        return $\mathbf{w}$
end
```

Syntax:

predicted_targets = Backpropagation_Stochastic(training_patterns, training_targets, test_patterns, input parameters);

[predicted_targets, Wih, Who] = Backpropagation_Stochastic(training_patterns, training_targets, test_patterns, input parameters);

[predicted_targets, Wih, Who, errors_throughout_training] = Backpropagation_Stochastic(training_patterns, training_targets, test_patterns, input parameters);

where**:**

Wih are the input-to-hidden unit weights

Who are the hidden-to-output unit weights

Input parameters:

1. The number of hidden units n_H.

2. The convergence criterion θ.

3. The convergence rate.

Additional outputs:

1. The input-to-hidden weights w_{ji}.

2. The hidden-to-output weights w_{kj}.

3. The test errors through the training.

Stochastic Backpropagation with momentum

Function name: `Backpropagation_SM`

Description:

This algorithm implements the stochastic backpropagation learning algorithm in a three-layer network of nonlinear units with momentum.

Pseudo-code:

```
begin initialize $n_H$, $\mathbf{w}$, $\alpha(<1)$, $\theta$, $\eta$, $m \leftarrow 0$,  $b_{ji} \leftarrow 0$,  $b_{kj} \leftarrow 0$
          do  $m \leftarrow m + 1$
                 $\mathbf{x}^m \leftarrow randomly\ chosen\ pattern$
                 $b_{ji} \leftarrow \eta(1-\alpha)\delta_j x_i + \alpha b_{ji}$ ;  $b_{kj} \leftarrow \eta(1-\alpha)\delta_k y_j + \alpha b_{kj}$
          until $\|\nabla J(\mathbf{w})\| < \theta$
     return $\mathbf{w}$
end
```

Syntax:

predicted_targets = Backpropagation_SM(training_patterns, training_targets, test_patterns, input parameters);

[predicted_targets, Wih, Who] = Backpropagation_SM(training_patterns, training_targets, test_patterns, input parameters);

[predicted_targets, Wih, Who, errors_throughout_training] = Backpropagation_SM(training_patterns, training_targets, test_patterns, input parameters);

where**:**

Wih are the input-to-hidden unit weights

Who are the hidden-to-output unit weights

Input parameters:

1. The number of hidden units n_H.

2. The convergence criterion θ.

3. The convergence rate.

Additional outputs:

1. The input-to-hidden weights w_{ji}.

2. The hidden-to-output weights w_{kj}.

3. The test errors through the training.

Batch Backpropagation

Function name: `Backpropagation_Batch`

Description:

This algorithm implements the batch backpropagation learning algorithm in a three-layer network of nonlinear units.

Pseudo-code:

```
begin initialize $n_H$, $\mathbf{w}$, criterion $\theta$, $\eta$, $r \leftarrow 0$
          do  $r \leftarrow r + 1$   (increment epoch)
               $m \leftarrow 0$ ; $\Delta w_{ji} \leftarrow 0$ ; $\Delta w_{kj} \leftarrow 0$
               do $m \leftarrow m + 1$
                    $\mathbf{x}^m \leftarrow select\ pattern$
                    $\Delta w_{ji} \leftarrow \Delta w_{ji} + \eta\delta_j x_i$ ; $\Delta w_{kj} \leftarrow \Delta w_{kj} + \eta\delta_k y_j$
               until $m = n$
               $w_{ji} \leftarrow w_{ji} + \eta\delta_j x_i$ ; $w_{kj} \leftarrow w_{kj} + \eta\delta_k y_j$
          until $\|\nabla J(\mathbf{w})\| < \theta$
     return $\mathbf{w}$
end
```

Syntax:

predicted_targets = Backpropagation_Batch(training_patterns, training_targets, test_patterns, input parameters);

[predicted_targets, Wih, Who] = Backpropagation_Batch(training_patterns, training_targets, test_patterns, input parameters);

[predicted_targets, Wih, Who, errors_throughout_training] = Backpropagation_Batch(training_patterns, training_targets, test_patterns, input parameters);

where**:**

Wih are the input-to-hidden unit weights

Who are the hidden-to-output unit weights

Input parameters:

1. The number of hidden units n_H.

2. The convergence criterion θ.

3. The convergence rate.

Additional outputs:

1. The input-to-hidden weights w_{ji}.

2. The hidden-to-output weights w_{kj}.

3. The training and test errors through the training.

Backpropagation trained using Conjugate Gradient Descent

Function name: `Backpropagation_CGD`

Description:

This algorithm trains a three-layer network of nonlinear units using conjugate gradient descent (CGD). CGD usually helps the network converge faster than first order methods.

Syntax:

predicted_targets = Backpropagation_CGD(training_patterns, training_targets, test_patterns, input parameters);

[predicted_targets, Wih, Who] = Backpropagation_CGD(training_patterns, training_targets, test_patterns, input parameters);

[predicted_targets, Wih, Who, errors_throughout_training] = Backpropagation_CGD(training_patterns, training_targets, test_patterns, input parameters);

where**:**

Wih are the input-to-hidden unit weights

Who are the hidden-to-output unit weights

Input parameters:

1. The number of hidden units, n_H.

2. The convergence criterion θ.

Additional outputs:

1. The input-to-hidden weights w_{ji}.

2. The hidden-to-output weights w_{kj}.

3. The training error through the training.

Recurrent Backpropagation

Function name: `Backpropagation_Recurrent`

Description: This algorithm trains a three-layer network of nonlinear units having recurrent connections. The network is fed with the inputs, and these are propagated until the network stabilizes. Then the weights are changed just as in traditional feed-forward networks.

Syntax:

```
predicted_targets = Backpropagation_Recurrent(training_patterns, training_targets, test_patterns, input parameters);

[predicted_targets, weights] = Backpropagation_Recurrent(training_patterns, training_targets, test_patterns,
                                                          input parameters);

[predicted_targets, weights, errors_throughout_training] = Backpropagation_Recurrent(training_patterns,
                                                          training_targets, test_patterns, input parameters);
```

Input parameters:

1. The number of hidden units, n_H.

2. The convergence criterion θ.

3. The convergence rate.

Additional outputs:

1. The connection weights.

2. The errors through the training.

Cascade-Correlation

Function name: `Cascade_Correlation`

Description: This algorithm trains a nonlinear cascade-correlation neural network.

Pseudo-code

```
begin initialize $\boldsymbol{a}$, criterion $\theta, \eta, k \leftarrow 0$
        do $m \leftarrow m + 1$
                $w_{ki} \leftarrow w_{ki} - \eta \nabla J(\mathbf{w})$
        until $\|\nabla J(\mathbf{w})\| < \theta$
        if $J(\boldsymbol{w}) > \theta$ then add hidden unit until exit
                do $m \leftarrow m + 1$
                $w_{ji} \leftarrow w_{ji} - \eta \nabla J(\mathbf{w})$ ; $w_{kj} \leftarrow w_{kj} - \eta \nabla J(\mathbf{w})$
        until $\|\nabla J(\mathbf{w})\| < \theta$
        return $\boldsymbol{w}$
end
```

Syntax:

predicted_targets = Cascade_Correlation(training_patterns, training_targets, test_patterns, input parameters);

[predicted_targets, Wih, Who] = Cascade_Correlation(training_patterns, training_targets, test_patterns, input parameters);

[predicted_targets, Wih, Who, errors_throughout_training] = Cascade_Correlation(training_patterns, training_targets, test_patterns, input parameters);

where**:**

Wih are the input-to-hidden unit weights

Who are the hidden-to-output unit weights

Input parameters:

1. The convergence criterion θ.

2. The convergence rate.

Additional outputs:

1. The input-to-hidden weights w_{ji}.

2. The hidden-to-output weights w_{kj}.

3. The training error through the training.

Optimal Brain Surgeon

Function name: `Optimal_Brain_Surgeon`

Description:

This algorithm prunes a trained three-layer network by means of Optimal Brain Surgeon or Optimal Brain Damage.

Pseudo-code:

```
begin initialize $n_H$, $\mathbf{a}$, $\theta$
        train a reasonably large network to minimum error
         do compute $\mathbf{H}^{-1}$ (inverse Hessian matrix)
                $q^* \leftarrow \arg min_q \frac{w_q^2}{2[\mathbf{H}^{-1}]_{qq}}$ (saliency $L_q$)
                $\mathbf{w} \leftarrow \mathbf{w} - \frac{w_{q^*}}{[H^{-1}]_{q^*q^*}} \mathbf{H}^{-1} \mathbf{e}_{q^*}$
         until $J(\mathbf{w}) > \theta$
        return $\mathbf{w}$
end
```

Syntax:

```
predicted_targets = Optimal_Brain_Surgeon(training_patterns, training_targets, test_patterns, input parameters);

[predicted_targets, Wih, Who] = Optimal_Brain_Surgeon(training_patterns, training_targets, test_patterns,
                                                                input parameters);

[predicted_targets, Wih, Who, errors_throughout_training] = Optimal_Brain_Surgeon(training_patterns,
                                                                training_targets, test_patterns, input parameters);
```

where**:**

Wih are the input-to-hidden unit weights

Who are the hidden-to-output unit weights

Input parameters:

1. The initial number of hidden units.

2. The convergence rate.

Additional outputs:

1. The input-to-hidden weights w_{ji}.

2. The hidden-to-output weights w_{kj}.

3. The training error through the training.

Quickprop

Function name: `Backpropagation_Quickprop`

Description:

This algorithm trains a three-layer network by means of the Quickprop algorithm.

Syntax:

```
predicted_targets = Backpropagation_Quickprop(training_patterns, training_targets, test_patterns, input parameters);

[predicted_targets, Wih, Who] = Backpropagation_Quickprop(training_patterns, training_targets, test_patterns,
                                                                    input parameters);

[predicted_targets, Wih, Who, errors_throughout_training] = Backpropagation_Quickprop(training_patterns,
                                                                    training_targets, test_patterns, input parameters);
```

where**:**
Wih are the input-to-hidden unit weights
Who are the hidden-to-output unit weights

Input parameters:

1. The number of hidden units n_H.

2. The convergence criterion.

3. The convergence rate.

4. The error correction rate.

Additional outputs:

1. The input-to-hidden weights w_{ji}.

2. The hidden-to-output weights w_{kj}.

3. The training error through the training.

Projection Pursuit

Function name: `Projection_Pursuit`

Description:

This algorithm implements the projection pursuit statistical estimation procedure.

Syntax:

```
predicted_targets = Projection_Pursuit(training_patterns, training_targets, test_patterns, input parameter);

[predicted_targets, component_weights, output_weights] = Projection_Pursuit(training_patterns, training_targets,
                                                                            test_patterns, input parameter);
```

Input parameters:

The number of component features onto which the data is projected.

Additional outputs:

1. The component weights.

2. The output unit weights

Radial Basis Function Classifier

Function name: `RBF_Network`

Description:

This algorithm trains a radial basis function classifier. First the algorithm computes the centers for the data using *k*-means. Then the algorithm estimates the variance of the data around each center, and uses this estimate to compute the activation of each training pattern to these centers. These activation patterns are used for computing the gating unit of the classifier, via the Moore-Penrose pseudo-inverse.

Syntax:

predicted_targets = RBF_Network(training_patterns, training_targets, test_patterns, input parameter);

[predicted_targets, component_weights, output_weights] = RBF_Network(training_patterns, training_targets, test_patterns, input parameter);

Input parameter:

The number of hidden units.

Additional outputs:

1. The locations in feature space of the centers of the hidden units.

2. The weights of the gating units.

Chapter 7

Stochastic Simulated Annealing

Function name: `Stochastic_SA`

Description: This algorithm clusters the patterns using stochastic simulated annealing in a network of binary units.

Pseudo-code:

```
begin initialize $T(k), k_{max}, s_i(1), w_{ij}$ for $i, j = 1, \ldots N$
      $k \leftarrow 0$
            do $k \leftarrow k + 1$
                  do select node $i$ randomly; suppose its state is $s_i$
                        $E_a \leftarrow -1/2 \sum_{j}^{N_i} w_{ij} s_i s_j$
                        $E_b \leftarrow -E_a$
                        if $E_b < E_a$
                              then $s_i \leftarrow -s_i$
                              else if $e^{-\frac{(E_b - E_a)}{T(k)}} > Rand[0, 1]$
                                    then $s_i \leftarrow -s_i$
                        until all nodes polled several times
                  until $k = k_{max}$ or stopping criterion met
      return $E, s_i$, for $i = 1, \ldots N$
```

end

Syntax:

```
[new_patterns, new_targets] = Stochastic_SA(training_patterns, training_targets, input_parameters, plot_on);
```

Input parameters:

1. The number of output data points.

2. The cooling rate.

The input flag `plot_on` determines if the algorithm’s progress should be shown through the learning iterations.

Deterministic Simulated Annealing

Function name: `Deterministic_SA`

Description:

This algorithm clusters the data using deterministic simulated annealing in a network of binary units.

Pseudo-code

```
begin initialize $T(k), w_{ij}, s_i(1)$ for $i, j = 1, \ldots N$
        $k \leftarrow 0$
                do $k \leftarrow k + 1$
                        select node $i$ randomly
                                $l_i \leftarrow \sum_j^{N_i} w_{ij} s_j$
                                $s_i \leftarrow f(l_i, T(k))$
                until $k = k_{max}$ or stopping criterion met
        return $E, s_i$, for $i = 1, \ldots N$
end
```

Syntax:

[new_patterns, new_targets] = Stochastic_SA(training_patterns, training_targets, input_parameters, plot_on);

Input parameters:

1. The number of output data points.

2. The cooling rate.

The input flag `plot_on` determines if the algorithm’s progress should be shown through the learning iterations.

Deterministic Boltzmann Learning

Function name: `BoltzmannLearning`

Description: Use deterministic Bolzmann learning to find a good combination of weak learners to classify data.

Pseudo-code

```
begin initialize D, $\eta$, $T(k)$, $w_{ij}$ for $i, j = 1, ... N$
          do randomly select training pattern $\boldsymbol{x}$
                    randomize states $s_i$
                    anneal network with input and output clamped
                    at final, low $T$, calculate $[s_i s_j]_{\alpha^i \alpha^o clamped}$
                    randomize states $s_i$
                    anneal network with input clamped but output free
                    at final, low $T$, calculate $[s_i s_j]_{\alpha^i clamped}$
                    $w_{ij} \leftarrow w_{ij} + (\eta / T)[[s_i s_j]_{\alpha^i \alpha^o clamped} - [s_i s_j]_{\alpha^i clamped}]$
                    until $k = k_{max}$ or stopping criterion met
          return $w_{ij}$
end
```

Syntax:

predicted_targets = Deterministic_Boltzmann(training_patterns, training_targets, test_patterns, input parameters);

[predicted_targets, updates_throughout_learning] = Deterministic_Boltzmann(training_patterns, training_targets, test_patterns, input parameters);

Input parameters:

1. The number of input units.

2. The number of hidden units.

3. The cooling rate.

4. The type of weak learner.

5. The parameters of the weak learner.

Additional outputs:

The errors during training.

Basic Genetic Algorithm

Function name: `Genetic_Algorithm`

Description:

This implementation uses a basic genetic algorithm to build a classifier from components of weak classifiers.

Pseudo-code

```
begin initialize $\theta$, $P_{co}$, $P_{mut}$, $L$  $N$-bit chromosomes
              do determine the fitness of each chromosome $f_i$, $i = 1,..., L$
                     rank the chromosomes
                     do select two chromosomes with the highest score
                            if Rand[0,1) < $P_{co}$ then crossover the pair at a randommly chosen bit
                                          else change each bit with proability $P_{mut}$;
                                                   remove the parent chromosomes
                     until $N$ offspring have been created
              until any chromosome's score $f$ exceeds $\theta$
       return highest fitness chromosome (best classifier)
end
```

Syntax:

```
predicted_targets = Genetic_Algorithm(training_patterns, training_targets, test_patterns, input parameters);
```

Input parameters:

1. The probability of cross-over P_{co}.

2. The probability of mutation P_{mut}.

3. The type of weak classifier.

4. The parameters of the weak learner.

5. The target or stopping error on training set.

6. The number of solutions to be returned by the program.

Genetic Programming

Function name: `Genetic_Programming`

Description: This algorithm approximates a function by evolving mathematical expressions by a genetic programming algorithm. The function is used to classify the data.

Syntax:

predicted_targets = Genetic_Programming(training_patterns, training_targets, test_patterns, input parameters);

[predicted_targets, best_function_found] = Genetic_Programming(training_patterns, training_targets, test_patterns, input parameters);

Input patterns:

1. The initial function length.

2. The number of generations.

3. The number of solutions to be returned by the program.

Additional outputs:

The best function found by the algorithm.

Chapter 8

C4.5

Function name: `C4_5`

Description:

Construct a decision tree recursively so as to minimize the error on a training set. Discrete features are split using a histogram and continuous features are split using an information criteria. The algorithm is implemented under the assumption that a pattern vector with fewer than 10 unique values is discrete, and will be treated as such. Other vectors are treated as continuous. Note that due to MATLAB memory and processing restrictions, the recursion depth may be reached during the processing of a large complicated data set, which will result in an error.

Syntax:

```
predicted_targets = C4_5(training_patterns, training_targets, test_patterns, input parameter);
```

Input parameter:

The maximum percentage of error at a node that will prevent it from further splitting.

CART

Function name: `CART`

Description:

Construct a decision tree recursively so as to minimize the error on a training set. The criterion for splitting a node is either the percentage of incorrectly classified samples at the node, or the entropy at the node, or the variance of the outputs. Note that due to MATLAB memoery and processing restrictions, the recursion depth may be reached during the processing of a large complicated data set, which will result in an error.

Syntax:

```
predicted_targets = CART(training_patterns, training_targets, test_patterns, input parameters);
```

Input parameters:

1. The splitting criterion (`entropy`, `variance`, or `misclassification`).

2. Maximum percentage of incorrectly assigned samples at a node.

ID3

Function name: `ID3`

Description:

Construct a decision tree recursively so as to minimize the error on a training set. This algorithm assumes that the data takes discrete values. The criterion for splitting a node is the percentage of incorrectly classified samples at the node. Note that due to MATLAB memoery and processing restrictions, the recursion depth may be reached during the processing of a large complicated data set, which will result in an error.

Syntax:

```
predicted_targets = ID3(training_patterns, training_targets, test_patterns, input parameters);
```

Input parameters:

1. Maximum number of values the data can take (i.e. the number of values that the data will be binned into).

2. Maximum percentage of incorrectly assigned samples at a node.

Naive String Matching

Function name: `Naive_String_Matching`

Description:

Perform naive string matching, which is quite inefficient in the general case. The value of this program is primarily for making performance comparisons with the Boyer-Moore algorithm. Note that this algorithm is in the “Other” directory.

Pseudo-code

```
begin initialize A, $\mathbf{a}$, $n \leftarrow \text{length}[text]$, $m \leftarrow \text{length}[\mathbf{x}]$
    $s \leftarrow 0$
        while $s < n - m$
            if $\mathbf{x}[1...m] = text[s+1...s+m]$
                then print “pattern occurs at shift” $s$
                    $s \leftarrow s + 1$
    return
end
```

Syntax:

location = Naive_String_Matching(text_vector, search_string);

Boyer-Moor String Matching

Function name: `Boyer_Moore_string_matching`

Description:

Perform string matching by the Boyer-Moore algorithm, which is typically far more efficient than naive string matching. Note that this algorithm is in the “Other” directory.

Pseudo-code

```
begin initialize A, a, n ← length[text], m ← length[x]
     F(x) ← last-occurence function, G(x) ← good-suffix function
          s ← 0
          while s ð n - m
               do j ← m
               while j > 0 and x[j] = text[s+j]
                    do j ← j – 1
                    if j = 0
                         then print “pattern occurs at shift” s
                            s ← s + G(0)
                         else s ← s + max[G(j), j – F(text[s + j])]
     return
end
```

Syntax:

location = Naive_String_Matching(text_vector, search_string);

Edit Distance

Function name: `Edit_Distance`

Description:

Compute the edit distance between two strings ***x*** and ***y***. Note that this algorithm is in the “Other” directory.

Pseudo-code

```
begin initialize A, $\boldsymbol{x}$, $\boldsymbol{y}$,   $m \leftarrow$ length[$\boldsymbol{x}$] ,   $n \leftarrow$ length[$\boldsymbol{y}$]
        $\boldsymbol{C}[0, 0] \leftarrow 0$
        $i \leftarrow 0$
        do $i \leftarrow i + 1$
                $\boldsymbol{C}[i, 0] \leftarrow i$
        until $i = m$
        $j \leftarrow 0$
        do $j \leftarrow j + 1$
                $\boldsymbol{C}[0, j] \leftarrow j$
        until $j = n$
        $i \leftarrow 0$ ; $j \leftarrow 0$
        do $i \leftarrow i + 1$
                do $j \leftarrow j + 1$
                        $\mathbf{C}[i,j] = \min[\mathbf{C}[i\text{-}1,j] + 1, \mathbf{C}[i,j\text{-}1]+1,\mathbf{C}[i\text{-}1,j\text{-}1]+ 1 - \delta(\mathbf{x}[i],\mathbf{y}[j])]$
                until $j = n$
```

```
          until $i = m$
          return C[$m,n$]
end
```

Syntax:

```
distance_matrix = Edit_Distance(text_vector1, text_vector2);
```

Bottom-Up Parsing

Function name: `Bottom_Up_Parsing`

Description:

Perform bottom-up parsing of a string ***x*** in grammar *G*. Note that this algorithm is in the “Other” directory.

Pseudo-code

```
begin initialize $G = ($A, I, S, P$)$, $\mathbf{x} = x_1x_2...x_n$
    $i \leftarrow 0$
    do $i \leftarrow i + 1$
        $V_{i1} \leftarrow \{A|A \to x_i\}$
    until $i = n$
    $j \leftarrow 1$
    do $j \leftarrow j + 1$
        $i \leftarrow 0$
        do $i \leftarrow i + 1$
            $V_{ij} \leftarrow \varnothing$
            $k \leftarrow 0$
            do $k \leftarrow k + 1$
                $V_{ij} \leftarrow V_{ij} \cup \{A|A \to BC \in P, B \in V_{ik} \text{and} C \in V_{i+k, j-k}\}$
            until $k = j - 1$
```

```
                  until $i = n - j + 1$
        until $j = n$
        if $S \in V_{1n}$ then print "parse of" $\boldsymbol{x}$ "successful in $G$"
    return
end
```

Syntax:

```
parsing_table = Bottom_Up_Parsing(alphabet_vector, variable_vector, root_symbol, production_rules, text_vector);
```

Grammatical Inference (Overview)

Function name: `Grammatical_Inference`

Description:

Infers a grammar *G* from a set of positive and negative example strings and a (simple) initial grammar G^0. Note that this algorithm is in the “Other” directory.

Pseudo-code

```
begin initialize $D^+$,$D^-$,$G^0$
    $n^\dagger \leftarrow |D^\dagger|$ (number of instances in $D^+$)
        $\mathrm{S} \leftarrow S$
        $\mathrm{A} \leftarrow$ set of characters in $D^+$
        $i \leftarrow 0$
        do $i \leftarrow i + 1$
            read $\boldsymbol{x}_i^+$ from $D^+$
            if $\boldsymbol{x}_i^+$ cannot be parsed by $G$
            then do propose additional productions to P and variables to I
                accept updates if $G$ parses $\boldsymbol{x}_i^+$ but no string in $D^-$
            until $i = n^+$
            eliminate redundant productions
    return $G \leftarrow \{\mathrm{A, I, S, P}\}$
end
```

Syntax:

```
[alphabet_vector, variable_vector, root_symbol, production_rules] = Grammatical_Inference
                                                                     (text_vectors_to_parse, labels);
```

Chapter 9

AdaBoost

Function name: `Ada_Boost`

Description:

AdaBoost builds a nonlinear classifier by constructing an ensemble of “weak” classifiers (i.e., ones that need perform only slightly better than chance) so that the joint decision is has better accuracy on the training set. It is possible to iteratively add classifiers so as to attain any given accuracy on the training set. In AdaBoost each sample of the training set is selected for training the weak with a probability proportional to how well it is classified. An incorrectly classified sample will be chosen more frequently for the training, and will thus be more likely to be correctly classified by the new weak classifier.

Pseudo-code

```
begin initialize D = {$\mathbf{x}^1, y_1, \ldots \mathbf{x}^n, y_n$}, $k_{max}$, $W_1(i) = 1/n, i = 1, \ldots, n$
    $k \leftarrow 0$
    do $k \leftarrow k + 1$
        train weak learner $C_k$ using D sampled according to $W_k(i)$
        $E_k \leftarrow$ training error of $C_k$ measured on D using $W_k(i)$
        $\alpha_k \leftarrow \frac{1}{2}\ln[(1 - E_k)/E_k]$
        $W_{k+1}(i) \leftarrow \frac{W_k(i)}{Z_k} \times \begin{cases} e^{-\alpha_k} if h_k(\mathbf{x}^i) = y_i \\ e^{\alpha_k} if h_k(\mathbf{x}^i) \neq y_i \end{cases}$
        until $k = k_{max}$
        **return** $C_k$ and $\alpha_k$ for $k = 1$ to $k_{max}$ (ensemble of classifiers with weights)
**end**
```

Syntax:

```
predicted_targets = Ada_Boost(training_patterns, training_targets, test_patterns, input parameters);

[predicted_targets, training_errors] = Ada_Boost(training_patterns, training_targets, test_patterns, input parameters);
```

Input parameters:

1. The number of boosting iterations.

2. The name of weak learner.

3. The parameters of the weak learner.

Additional outputs:

The training errors throughout the learning.

Local boosting

Function name: `LocBoost`

Description:

Create a single nonlinear classifier based on boosting of localized classifiers. The algorithm assigns local classifiers to incorrectly classified training data, and optimizes these local classifiers to reach the minimum error.

Syntax:

```
predicted_targets = LocBoost(training_patterns, training_targets, test_patterns, input parameters);
```

Input parameters:

1. The number of boosting iterations.

2. The number of EM iterations.

3. The number of optimization steps.

4. The type of weak learner.

5. The weak learner parameters.

Reference

R. Meir, R. El-Yaniv and S. Ben-David, "Localized boosting," *Proceedings of the 13th Annual Conference on Computational Learning Theory*

Bayesian Model Comparison

Function name: `Bayesian_Model_Comparison`

Description:

Bayesian model comparison, as implemented here, selects the best mixture of Gaussians model for the data. Each full candidate model is constructed using Expectation-Maximization. The program then computes the Occam factor and finally returns the model that maximizes the Occam factor.

Syntax:

```
predicted_targets = Bayesian_Model_Comparison(training_patterns, training_targets, test_patterns, input parameters);
```

Input parameters: Maximum number of Gaussians for each models.

Component Classifiers with Discriminant Functions

Function name: `Components_with_DF`

Description:

This implementation uses logistic component classifiers and a softmax gating function to create a global classifier. The parameters of the components are learned using Newton descent, and the parameters of the gating system using gradient descent.

Syntax:

predicted_targets = Components_with_DF(training_patterns, training_targets, test_patterns, input parameters);

[predicted_targets, errors] = Components_with_DF(training_patterns, training_targets, test_patterns, input parameters);

Input parameters:

The component classifiers as pairs of classifier name and classifier parameters.

Additional outputs:

The errors through the training.

Component Classifiers without Discriminant Functions

Function name: `Components_without_DF`

Description: This program works with any of the classifiers in the toolbox as components to build a single meta-classifier. The gating unit parameters are learned through gradient descent.

Syntax:

predicted_targets = Components_without_DF(training_patterns, training_targets, test_patterns, input parameters);

[predicted_targets, errors] = Components_without_DF(training_patterns, training_targets, test_patterns, input parameters);

Input parameters:

The component classifiers as pairs of classifier name and classifier parameters.

Additional outputs:

The errors through the training.

ML_II

Function name: `ML_II`

Description:

This algorithm finds the best multiple Gaussian model for the data, and uses this model to construct a decision surface. The algorithm computes the Gaussian parameters for the data via the EM algoritm, assuming varying number of Gaussians. Then, the algorithm computes the probability that the data was generated by these models and returns the most likely such model. Finally, the algorithm uses the parameters of this model to construct the Bayes decision region.

Syntax:

```
predicted_targets = ML_II(training_patterns, training_targets, test_patterns, input parameter);
```

Input parameters:

Maximum number of Gaussians components per class.

Interactive Learning

Function name: `Interactive_Learning`

Description:

This algorithm implements interactive learning in a particular type of classifer, specifically, the nearest-neighbor interpolation on the training data. The training points that have the highest ambiguity are referred to the user for labeling, and each such label is used for improving the classification.

Syntax:

```
predicted_targets = Interactive_Learning(training_patterns, training_targets, test_patterns, input parameters);
```

Input parameters:

1. The number of points presented as queries to the user.

2. The weight of each queried point relative the other data points.

Chapter 10

k-Means Clustering

Function name: `K_means`

Description:

This is a top-down clustering algorithm which attempts to find the *c* representative centers for the data. The initial means are selected from the training data itself. k-means is biased towards spherical clusters with similar variances.

Pseudo-code

```
begin initialize $n, c, \mu_1, \mu_2, ..., \mu_c$
    do classify $n$ samples according to nearest $\mu_i$
        recompute $\mu_i$
        until no change in $\mu_i$
    return $\mu_1, \mu_2, ..., \mu_c$
end
```

Syntax:

```
[clusters, cluster_labels] = k_means(patterns, targets, input_parameter, plot_on);

[clusters, cluster_labels, original_data_labels] = k_means(patterns, targets, input_parameter, plot_on);
```

Input parameter:

The number of desired output clusters, *c*.

The input parameter `plot_on` determines if the cluster centers are plotted during training.

Additional outputs:

The number of the cluster assigned to each input pattern.

Fuzzy *k*-Means Clustering

Function name: `fuzzy_k_means`

Description:

This is a top-down clustering algorithm which attempts to find the *k* representative centers for the data. The initial means are selected from the training data itself. This algorithm uses a slightly different gradient search than the simple standard k-means algorithm, but generally yields the same final solution.

Pseudo-code

```
begin initialize $n, c, b, \mu_1, \mu_2, ..., \mu_c,, \hat{P}(\omega_i|\mathbf{x}_j)$ , $i = 1, ..., c; j = 1, ..., n$
          normalize $\hat{P}(\omega_i|\mathbf{x}_j)$
          do recompute $\mu_i$
               recompute $\hat{P}(\omega_i|\mathbf{x}_j)$
          until small change in $\mu_1$ and $\hat{P}(\omega_i|\mathbf{x}_j)$
     return $\mu_1, \mu_2, ..., \mu_c$
end
```

Syntax:

```
[clusters, cluster_labels] = fuzzy_k_means(patterns, targets, input_parameter, plot_on);

[clusters, cluster_labels, original_data_labels] = fuzzy_k_means(patterns, targets, input_parameter, plot_on);
```

Input parameter:

The number of desired output clusters, *c*.

The input parameter `plot_on` determines if the cluster centers are plotted during training.

Additional outputs:

The number of the cluster assigned to each input pattern.

Kernel *k*-Means Clustering

Function name: `kernel_k_means`

Description:

This is a top-down clustering algorithm which is identical to the k-means algorithm (See above), except that the data is first mapped to a new space using a kernel function.

Syntax:

```
[clusters, cluster_labels] = kernel_k_means(patterns, targets, input_parameter, plot_on);

[clusters, cluster_labels, original_data_labels] = kernel_k_means(patterns, targets, input_parameter, plot_on);
```

Input parameters:

1. The number of desired output clusters, *c*.
2. The kernel function: `Gauss` (or `RBF`), `Poly`, `Sigmoid`, or `Linear`.
3. Kernel parameter: For each kernel parameters the following parameters are needed:

- RBF kernel: Gaussian width (scalar parameter)
- Poly kernel: The integer degree of the polynomial
- Sigmoid: The slope and constant of the sigmoid
- Linear: no parameters are needed

The input parameter `plot_on` determines if the cluster centers are plotted during training.

Additional outputs:

The number of the cluster assigned to each input pattern.

Spectral *k*-Means Clustering

Function name: `spectral_k_means`

Description:

This is a top-down clustering algorithm which is identical to the k-means algorithm (See above), except that the data is first mapped to a new space using a kernel function and the clustering is performed in that space.

Syntax:

```
[clusters, cluster_labels] = spectral_k_means(patterns, targets, input_parameter, plot_on);

[clusters, cluster_labels, original_data_labels] = spectral_k_means(patterns, targets, input_parameter, plot_on);
```

Input parameters:

1. The number of desired output clusters, *c*.
2. The kernel function: `Gauss` (or `RBF`), `Poly`, `Sigmoid`, or `Linear`.
3. Kernel parameter: For each kernel parameters the following parameters are needed:

- RBF kernel: Gaussian width (scalar parameter)
- Poly kernel: The integer degree of the polynomial
- Sigmoid: The slope and constant of the sigmoid
- Linear: no parameters are needed

4. Clutering type: The clustering type can be:

- Multicut
- NJW (According to the method proposed by Ng, Jordan, and Weiss)

The input parameter `plot_on` determines if the cluster centers are plotted during training.

Additional outputs:
The number of the cluster assigned to each input pattern.

Basic Iterative Minimum-Squared-Error Clustering

Function name: `BIMSEC`

Description:

This algorithm iteratively searches for the c clusters that minimize the sum-squared error of the training data with respect to the nearest cluster center. The initial clusters are selected from the data itself.

Pseudo-code:

```
begin initialize $n, c, \boldsymbol{m}_1, \boldsymbol{m}_2, \ldots, \boldsymbol{m}_c$
        do randomly select a sample $\hat{\boldsymbol{x}}$
                $i \leftarrow \arg min_i \|\boldsymbol{m}_{i'} - \hat{\boldsymbol{x}}\|$ (classify $\hat{\boldsymbol{x}}$)
                        if $n_i$ ¦ $1$ then compute
                        $\rho_j = \begin{cases} \frac{n_j}{n_j + 1}\|\hat{\boldsymbol{x}} - \boldsymbol{m}_j\|^2 & j \neq i \\ \frac{n_j}{n_j - 1}\|\hat{\boldsymbol{x}} - \boldsymbol{m}_j\|^2 & j = i \end{cases}$
                if $\rho_k < \rho_j$ for all $j$ then transfer $\hat{\boldsymbol{x}}$ to $D_k$
                        recompute $J_e, \boldsymbol{m}_i, \boldsymbol{m}_k$
                until no change in $J_e$ in $n$ attempts
 return $\boldsymbol{m}_1, \boldsymbol{m}_{2i}, \ldots, \boldsymbol{m}_c$
end
```

Syntax:

```
[clusters, cluster_labels] = BIMSEC(patterns, targets, input_parameter, plot_on);

[clusters, cluster_labels, original_data_labels] = BIMSEC(patterns, targets, input_parameter, plot_on);
```

Input parameter:

The number of desired output clusters, *c*.

The input parameter `plot_on` determines if the cluster centers are plotted during training.

Additional outputs:

The number of the cluster assigned to each input pattern.

Agglomerative Hierarchical Clustering

Function name: AGHC

Description:

This function implements the bottom-up clustering. The algorithm starts by assuming each training point is its own cluster and then iteratively merges the nearest such clusters (where proximity is computed by a distance function) until the desired number of clusters are formed.

Pseudo-code

```
begin initialize $c$, $\hat{c} \leftarrow n$ , $D_i \leftarrow \{\boldsymbol{x}_i\}$ , $i = 1, ... n$
            do $\hat{c} \leftarrow \hat{c} - 1$
                  find nearest clusters, say, $D_i$ and $D_j$
                  merge $D_i$ and $D_j$
            until $c = \hat{c}$
      return $c$ clusters
end
```

Syntax:

```
[clusters, cluster_labels] = AGHC(patterns, targets, input_parameters, plot_on);

[clusters, cluster_labels, original_data_labels] = AGHC(patterns, targets, input_parameters, plot_on);
```

Input parameters:

1. The number of desired output clusters, *c*.

2. The type of distance function to be used (`min`, `max`, `avg`, or `mean`).

The input parameter `plot_on` determines if the cluster centers are plotted during training.

Additional outputs:

The number of the cluster assigned to each input pattern.

Stepwise Optimal Hierarchical Clustering

Function name: `SOHC`

Description:

This function implements the bottom-up clustering. The algorithm starts by assuming each training point is its own cluster and then iteratively merges the two clusters that change a clustering criterion the least, until the desired number of clusters *c* are formed.

Pseudo-code:

```
begin initialize $c, \hat{c} \leftarrow n$ , $D_i \leftarrow \{\mathbf{x}_i\}$ , $i = 1, \ldots n$
          do $\hat{c} \leftarrow \hat{c} - 1$
               find clusters whose merger changes the criterion the least, say, $D_i$ and $D_j$
               merge $D_i$ and $D_j$
          until $c = \hat{c}$
     return $c$ clusters
end
```

Syntax:

```
[clusters, cluster_labels] = SOHC(patterns, targets, input_parameter, plot_on);

[clusters, cluster_labels, original_data_labels] = SOHC(patterns, targets, input_parameter, plot_on);
```

Input parameter:

The number of desired output clusters, *c*.

The input parameter `plot_on` determines if the cluster centers are plotted during training.

Additional outputs:

The number of the cluster assigned to each input pattern.

Competitive Learning

Function name: `Competitive_learning`

Description:

This function implements competitive learning clustering, where the nearest cluster center is updated according to the position of a randomly selected training pattern.

Pseudo-code

```
begin initialize $\eta, n, c, k, \mathbf{w}_1, \ldots, \mathbf{w}_c$
            $\mathbf{x}_i \leftarrow \{1, \mathbf{x}_i\}$ , $i = 1, \ldots, n$   (augment all patterns)
            $\mathbf{x}_i \leftarrow \mathbf{x}_i / \|\mathbf{x}_i\|$ , $i = 1, \ldots, n$   (normalize all patterns)
             $j \leftarrow \arg max_{j'} \mathbf{w}_{j'}^t \mathbf{x}$      (classify $\mathbf{x}$)
            $\mathbf{w}_j \leftarrow \mathbf{w}_j + \eta \mathbf{x}$      (weight update)
             $\mathbf{w}_j \leftarrow \mathbf{w}_j / \|\mathbf{w}_j\|$     (weight normalization)
            until no significant change in $\mathbf{w}$ in $k$ attempts
      return $\mathbf{w}_1, \ldots, \mathbf{w}_c$
end
```

Syntax:

[clusters, cluster_labels] = Competitive_Learning(patterns, targets, input_parameters, plot_on);

[clusters, cluster_labels, original_data_labels] = Competitive_Learning(patterns, targets, input_parameters, plot_on);

[clusters, cluster_labels, original_data_labels, weights] = Competitive_Learning(patterns, targets, input_parameters, plot_on);

Input parameters:

1. The number of desired output clusters, *c*.

2. The learning rate.

The input parameter `plot_on` determines if the cluster centers are plotted during training.

Additional outputs:

1. The number of the cluster assigned to each input pattern.

2. The weight matrix representing the cluster centers.

Basic Leader-Follower Clustering

Function name: `Leader_Follower`

Description:

This function implements basic leader-follower clustering, which is similar to competitive learning but additionally generates a new cluster center whenever a new input pattern differs by more than a threshold distance θ from existing clusters.

Pseudo-code

```
begin initialize $\eta, \theta$
        $\mathbf{w}_1 \leftarrow \mathbf{x}$
        do accept new $\mathbf{x}$
          $j \leftarrow \arg max_{j'} \|\mathbf{x} - \mathbf{w}_{j'}\|$        (find nearest cluster)
            if $\|\mathbf{x} - \mathbf{w}_j\| < \theta$
                then $\mathbf{w}_j \leftarrow \mathbf{w}_j + \eta\mathbf{x}$
                else add new $\mathbf{w} \leftarrow \mathbf{x}$
            $\mathbf{w} \leftarrow \mathbf{w} / \|\mathbf{w}\|$        (normalize weight)
        until no more patterns
    return $\mathbf{w}_1, \mathbf{w}_2, \ldots$
end
```

Syntax:

```
[clusters, cluster_labels] = Leader_Follower(patterns, targets, input_parameters, plot_on);

[clusters, cluster_labels, original_data_labels] = Leader_Follower(patterns, targets, input_parameters, plot_on);

[clusters, cluster_labels, original_data_labels, weights] = Leader_Follower(patterns, targets, input_parameters,
                                                                                          plot_on);
```

Input parameters:

1. The minimum distance to connect across θ.

2. The rate of convergence.

The input parameter `plot_on` determines if the cluster centers are plotted during training.

Additional outputs:

1. The number of the cluster assigned to each input pattern.

2. The weight matrix representing the cluster centers.

Hierarchical Dimensionality Reduction

Function name: `HDR`

Description: This function clusters similar *features* so as to reduce the dimensionality of the data.

Pseudo-code:

```
begin initialize $d'$, $D_i \leftarrow \{\mathbf{x}_i\}$ , $i = 1, ..., d$
              $\hat{d} \leftarrow d + 1$
              do $\hat{d} \leftarrow \hat{d} - 1$
                   computer $\mathbf{R}$
                   find most correlated distinct clusters, say $D_i$ and $D_j$
                   $D_i \leftarrow D_i \cup D_j$      (merge)
                   delete $D_j$
              until $\hat{d} = d'$
         return $d'$ clusters
end
```

Syntax:

[new_patterns, new_targets] = HDR(patterns, targets, input_parameter);

Input parameter:

The desired number of dimensions d' for representing the data.

Independent Component Analysis

Function name: `ICA`

Description: Independent component analysis is a method for blind separation of signals. This method assumes there are *N* independent sources, linearly mixed to generate *M* signals, *MŠN*. The goal of this method is to find the mixing matrix that will make it possible to recover the source signals. The mixing matrix does not generate orthogonal sources (as in PCA), rather the sources are found so that they are as independent as possible. The program works in two stages. First, the data is standardized, i.e., whitened and scaled to the range *[-1, 1]*. The data is then rotated to find the correct mixing matrix; this rotation is performed via a nonlinear activation function. Possible functions are, for example, odd powers of the input and hyperbolic tangents.

Syntax:

```
[new_patterns, new_targets] = ICA(patterns, targets, input_parameters);

[new_patterns, new_targets, unmixing_mat] = ICA(patterns, targets, input_parameters);

[new_patterns, new_targets, unmixing_mat, reshaping matrix, means_vector] = ICA(patterns, targets,
                                                                        input_parameters);
```

Input parameters:

1. The output dimension.

2. The convergence rate.

Additional outputs:

1. The mixing matrix.

2. The unmixing matrix and the means of the inputs.

Online Single-Pass Clustering

Function name: `ADDC`

Description:

An on-line (single-pass) clustering algorithm which accepts a single sample at each step, updates the cluster centers and generates new centers as needed. The algorithm is efficient in that it generates the cluster centers with a single pass of the data.

Syntax:

```
[cluster_centers, cluster_targets] = ADDC(patterns, targets, input_parameter, plot_on);
```

Input parameter:

The number of desired clusters.

The input parameter `plot_on` determines if the cluster centers are plotted during training.

Reference:

I. D. Guedalia, M. London and M. Werman, "An on-line agglomerative clustering method for nonstationary data," *Neural Computation*, **11**:521-40 (1999).

Discriminant-Sensitive Learning Vector Quantization

Function name: `DSLVQ`

Description:

This function performs learning vector quantization (i.e., represents a data set by a small number of cluster centers) using a distinction or classification criterion rather than a traditional sum-squared-error criterion.

Syntax:

```
[new_patterns, new_targets] = DSLVQ(patterns, targets, input_parameter, plot_on);

[new_patterns, new_targets, weights] = DSLVQ(patterns, targets, input_parameter, plot_on);
```

Input parameter:

The number of desired output clusters, *c*.

The input parameter `plot_on` determines if the cluster centers are plotted during training.

Additional outputs:

The final weight vectors representing cluster centers.

Reference

M. Pregenzer, D. Flotzinger and G. Pfurtscheler, "Distinction sensitive learning vector quantization: A new noise-insensitive classification method," *Proceedings of the 4th International Conference on Artificial Neural Networks*, Cambridge UK (1995)

Exhaustive Feature Selection

Function name: `Exhaustive_Feature_Selection`

Description:

This function searches for the combination of features that yields the best classification accuracy on a data set. The search is exhaustive in subsets of features, and each subset is tested using 5-fold cross-validation on a given classifier. Note that applying this function when there are more than 10 features is impractical.

Syntax:

```
[new_patterns, new_targets] = Exhaustive_Feature_Selection(patterns, targets, input_parameters);

[new_patterns, new_targets, feature_numbers] = Exhaustive_Feature_Selection(patterns, targets, input_parameters);
```

Input parameters:

1. The output dimension.

2. The classifier type.

3. The parameters appropriate to the chosen classifier.

Additional outputs:

The indexes of the selected features.

Information-Based Feature Selection

Function name: `Information_based_selection`

Description:

This function selects the best features for classification based on information-theoretic considerations; the algorithm can be applied to virtually any basic classifier. However this program is often slow because the cross-entropy between *each* pair of features must be computed. Moreover, the program may be inaccurate if the number of data points is small.

Syntax:

```
[new_patterns, new_targets] = Information_based_selection(patterns, targets, input_parameter);

[new_patterns, new_targets, feature_numbers] = Information_based_selection(patterns, targets, input_parameter);
```

Input parameter:

The desired number of ouput dimensions.

Additional outputs:

The indexes of the features returned.

Reference:

D. Koller and M. Sahami, "Toward optimal feature selection," *Proceedings of the 13th International Conference on Machine Learning*, pp. 284-92 (1996)

Kohonen Self-Organizing Feature Map

Function name: `Kohonen_SOFM`

Description:

This function clusters the data by generating a self-organized feature map or "topologically correct map."

Syntax:

```
[clusters, cluster_labels] = Kohonen_SOFM(patterns, targets, input_parameters, plot_on);

[clusters, cluster_labels, original_data_labels] = Kohonen_SOFM(patterns, targets, input_parameters, plot_on);
```

Input parameter:

1. The number of desired output clusters, *c*.

2. Window width.

The input parameter `plot_on` determines if the cluster centers are plotted during training.

Additional outputs:

The number of the cluster assigned to each input pattern.

Multidimensional Scaling

Function name: `MDS`

Description:

This function represents a data set in a lower dimensional space such that if two patterns x_1 and x_2 are close in the original space, then their images y_1 and y_2 in the final space are also close. Conversely, if two patterns x_1 and x_3 are far apart in the initial space, then their images y_1 and y_3 in the final space are also far apart. The algorithm seeks an optimum of a global criterion function chosen by the user.

Syntax:

```
[clusters, cluster_labels] = MDS(patterns, targets, input_parameters);
```

Input parameters:

1. The criterion function J_{ee}, J_{ef}, or J_{ff} (`ee` - emphasize errors, `ef` - emphasize large products of errors and fractional errors, or `ff` - emphasize large fractional errors).

2. The number of output dimensions.

3. The convergence rate.

Minimum Spanning Tree Clustering

Function name: `min_spanning_tree`

Description:

This function builds a minimum spanning tree for a data set based on either nearest neighbors or inconsistent edges.

Syntax:

```
[clusters, cluster_labels] = min_spanning_tree(patterns, targets, input_parameters, plot_on);
```

Input parameter:

1. The linkage determination method (`NN` - nearest neighbor, `inc` - inconsistant edge).

2. The number of output data points per cluster or difference factor.

The input parameter `plot_on` determines if the cluster centers are plotted during training.

Principle Component Analysis

Function name: PCA

Description:

This function implements principle component analysis. First the algorithm subtracts the sample mean from each data point. Then the program computes the eigenvectors of the covariance matrix of the data and selects the largest eigenvalues and associated eigenvectors. The data is then transformed to a new hyperspace by multiplying them with these eigenvectors.

Syntax:

```
[new_patterns, new_targets] = PCA(patterns, targets, input_parameter);

[new_patterns, new_targets, unmixing_mat] = PCA(patterns, targets, input_parameter);

[new_patterns, new_targets, unmixing_mat, reshaping matrix, means_vector] = PCA(patterns, targets,
                                                                         input_parameter);
```

Input parameter:

The output dimension.

Additional outputs:

1. The mixing matrix.

2. The unmixing matrix and the means of the inputs.

Nonlinear Principle Component Analysis

Function name: `NLPCA`

Description:

The function implements a neural network with three hidden layers: a central layer of linear units and two nonlinear sigmoidal hidden layers. The number of units in the central linear layer is set equal to the desired output dimension. The network is trained as an auto-associator—i.e., mapping input to the same target input—and the nonlinear principle components are represented at the central linear layer.

Syntax:

```
[new_patterns, new_targets] = NLCA(patterns, targets, input_parameters);
```

Input parameters:

1. The number of desired output dimensions.

2. The number of hidden units in the nonlinear hidden layers.

Kernel Principle Component Analysis

Function name: `kernel_PCA`

Description:

This function implements principle component analysis with kernel functions. This algorithm is identical to principle component analysis, except that the data is first mapped to a new space using a kernel function

Syntax:

```
[new_patterns, new_targets] = kernel_PCA(patterns, targets, input_parameter);
```

Input parameters:

1. The output dimension.
2. The kernel function: `Gauss` (or `RBF`), `Poly`, `Sigmoid`, or `Linear`.
3. Kernel parameter: For each kernel parameters the following parameters are needed:

- RBF kernel: Gaussian width (scalar parameter)
- Poly kernel: The integer degree of the polynomial
- Sigmoid: The slope and constant of the sigmoid
- Linear: no parameters are needed

Linear Vector Quantization 1

Function name: `LVQ1`

Description:

This function finds a representative cluster centers for labeled data, and can thus be used as a clustering or as a classification method. The program moves cluster centers toward patterns that are in the same class as the centers, and moves other centers away from those patterns of other classes.

Syntax:

```
[clusters, cluster_labels] = LVQ1(patterns, targets, input_parameters, plot_on);
```

Input parameter:

The number of output data points.

The input parameter `plot_on` determines if the cluster centers are plotted during training.

Linear Vector Quantization 3

Function name: `LVQ3`

Description:

This function finds a representative cluster centers for labeled data, and can thus be used as a clustering or as a classification method. The program moves cluster centers toward patterns that are in the same class as the centers, and moves other centers away from those patterns of other classes. LVQ3 differs from LVQ1 in details of the rate of the weight updates.

Syntax:

```
[clusters, cluster_labels] = LVQ3(patterns, targets, input_parameters, plot_on);
```

Input parameter:

The number of output data points.

The input parameter `plot_on` determines if the cluster centers are plotted during training.

Sequential Feature Selection

Function name: `Sequential_Feature_Selection`

Description:

This function sequentially selects features for the lowest classification error. Then, until enough features are found, a feature that gives the largest reduction in classification error is added to the set. For backward selection, the process begins with the full set of features, and one is removed at each iteration.

Syntax:

```
[new_patterns, new_targets] = Sequential_Feature_Selection(patterns, targets, input_parameters);

[new_patterns, new_targets, feature_numbers] = Sequential_Feature_Selection(patterns, targets, input_parameters);
```

Input parameters:

1. The choice of search (`Forward` or `Backward`).

2. The output dimension.

3. The classifier type.

4. The parameters appropriate to the chosen classifier.

Additional outputs:

The indexes of the selected features.

Genetic Culling of Features

Function name: `Genetic_Culling`

Description:

This function performs feature selection using a genetic algorithm of the culling type, i.e., it selects subsets. The algorithm randomly partitions the features into groups of size N_g. Each candidate partition is evaluated for classification accuracy using five-fold cross validation. Then, the algorithm deletes a fraction of the worst-performing groups and generate the same number of groups by sampling from the remaining groups. The whole process then iterates until a criterion classification performance has been achieved or there is negligable improvement.

Syntax:

[new_patterns, new_targets] = Sequential_Feature_Selection(patterns, targets, input_parameters);

[new_patterns, new_targets, feature_numbers] = Sequential_Feature_Selection(patterns, targets, input_parameters);

Input parameters:

1. The fraction of groups discarded at each iteration.

2. The number of features in each solution (The output dimension).

3. The classifier type.

4. The parameters appropriate to the chosen classifier.

Additional outputs:

The indexes of the selected features.

Reference:

E. Yom-Tov and G. F. Inbar, "Selection of relevant features for classification of movements from single movement-related potentials using a genetic algorithm," *23rd Annual International Conference of the IEEE Engineering in Medicine and Biology Society* (2001).

References

1 R. O. Duda, P. E. Hart and D. G. Stork, **Pattern Classification** (2nd ed.), Wiley (2001)

2 The MathWorks, Inc., **MATLAB: The Language of Technical Computing**, The MathWorks, Inc. (2003)

3 K. Rose, "Deterministic annealing for clustering, compression, classification, regression, and related optimization problems, " *Proceedings of the IEEE*, **86**(1):2210-39 (1998)

4 K.-R. Müller, S. Mika, G. Rätsch, K. Tsuda and B. Schölkopf, "An introduction to kernel-based learning algorithms," *IEEE Transaction on Neural Networks*, **12**(2): 181-201 (2001)

Index