EE 8591: DEVELOPMENT AND EVALUATION OF LEARNING SOFTWARES

REPORT PREPARED BY

SAUPTIK DHAR
email: dharx007@umn.edu
Ph No: - (612)876-6429

GUIDED BY

PROFESSOR VLADIMIR CHERKASSKY
DEPARTMENT OF ECE
UNIVERSITY OF MINNESOTA

ACKNOWLEDGEMENTS

The culmination of the project to the present state is not just the single handed effort by the Author. The author would like to thank all who participated to bring the project to its present state. However the author would like to sincerely express his gratitude towards:

- Professor Vladimir Cherkassky: For guiding through the entire stages of the project. Moreover, the complete understanding of the regression interface would have been vacant without the guidance of Professor Cherkassky.

- Dr Ridder and Dr. David M.J Tax: For their guidance on the usage of the PRTools.

- Dr. Yunqian Ma: For his guidance and comments on the Project document.

- Predictive Learning Research Group: For helping in the implementation of the different new interfaces and highlighting the usage of some packages.

- Students of EE 8591 and Ms. Amisha Patel: For their useful comments on the XTAL package.

CONTENTS

INTRODUCTION

MOTIVATION

The main motivation behind the project is to provide a comparison between the available soft wares and to develop some useful interfaces for the usability of the existing packages. However in view of the huge repository of the available soft wares we limit ourselves mainly to the Software packages in the course website and some popular Machine learning software viz, WEKA, PRTools etc. We further try to provide some recommendations for the usage of software packages based on their usage.

PERFORMANCE METRICS

We analyze the different software mainly on the basis of the following Performance Metric:

1. User Interface of the Software package.
2. Parameter Tuning.
3. Output Parameters
4. Inherent Algorithm used.(a reference)
5. Help Documents.(a reference)

Note: Some of the packages used here are still not available publicly. Moreover some of the packages used are the unlicensed version of the original package. This is likely to produce a bias in the generated results.

PACKAGE UTILITY

We identify a basic package utility as one of the following type.

1. Clustering/Dimensionality Reduction package.
2. Classification package.
3. Regression package.
4. Application Specific package (we provide just one such software).

ORGANIZATION OF THE REPORT

The Report has been organized mainly in three sections. In the first section of the report we provide a detailed description of the usage of the SOTWARE PACKAGES that are available in the course website: http://www.ece.umn.edu/users/cherkass/ee8591/. In the second section we provide the other existing software packages and a brief introduction of the framework of their usage. In the third section we provide some comparison and recommendations for different kinds of software usages. Finally we provide some directions for improving the report to make it appropriate for general reference.

SECTION 1: COURSE WEBSITE

1.1. CLUSTERING/DIMENSIONALITY REDUCTION

We mainly provide the analysis of two of the most prevalent methods used in practice.

1. **Generalized Lloyd Algorithm.(GLA)**
2. **Self Organizing Maps (SOM).**

1. **Generalized Lloyd Algorithm:** This algorithm is used for clustering. For details of the algorithm please refer [1]. The implementation of the algorithm is available in the STPRTool package used in the course website [2]. The package is publicly available at: http://cmp.felk.cvut.cz/cmp/software/stprtool/index.html

   USER INTERFACE:
   The STPRTool provides mainly one interface for the (GLA).
   [model,labels] = cmeans(data,num_centers,Init_centers)

   - **INPUT PARAMETERS**
     The input parameters for the interface are:

     data: This is a data structure that contains the input training data. The data structure has mainly 4 parameters.

     data.X= a **DxN** array of the independent input variables.
     Here,
     D=Dimension of the input data.
     N=Number of Samples.

     data.num_data= Total number of samples

     data.dim= Input data Dimension

   - **TUNABLE PARAMETERS**

num_centers= Total number of centers K. This is a scalar value indicating the total number of centers.[default=1]

init_centers: Though it is specified in the help documents**. This parameter is not used internally!** (Awaiting an email response from Mr. Franc).

- **OUTPUT PARAMETERS**

model: This is a data structure which represents the clustering model. It has the following parameters

model .X : The cluster centers. This is an array of dimension: [num_centers x D]

Where,

num_centers=Number of centers provided by the user.

D= Dimension of the input samples.

model.y: Implicitly added labels 1..num_centers. This is used to identify the cluster to which a center belongs. This parameter is not for user's interpretation. This is mainly used by the **pboundary** interface to identify the cluster labels.

model.t : Total number of iterations for finding the model. This is a scalar value

model.MsErr: Mean-Square error at each iteration. This is an array of dimension [1xt].

where,

t=Total number of iterations for finding the model.

labels: This is the cluster label assigned to the input data. Labels assigned to data according to the nearest center. This output is used while assigning color map to the input samples using the ppatterns interface. This is an array of dimension [1 x num_data].

EXAMPLE (GLA ALGORITHM)

Here we illustrate the utility of the package using doughnut distribution. [1] (see pg 188)

Step 1: Generate the data

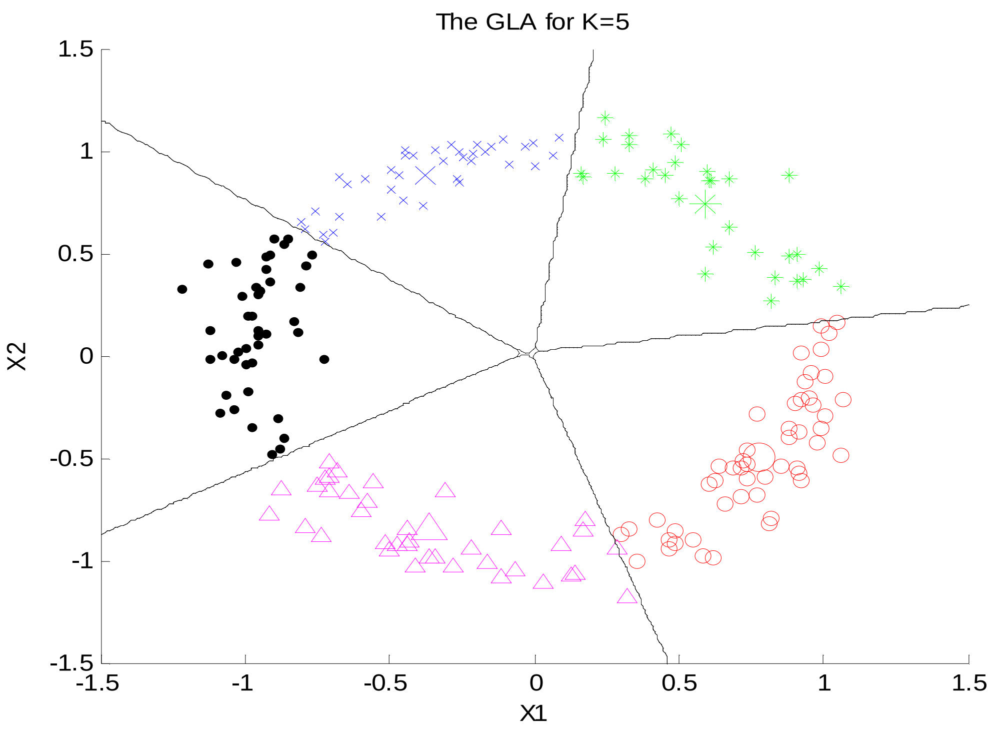
```
z=sort(10*rand(200,1));
noise=normrnd(0,0.1,200,2);                    %Bivariate Gaussian Noise with σ=0.1
traindata=[cos(2*pi*z) sin(2*pi*z)]+noise;            %Doughnut Distribution
data.X=traindata';                             %Form the input data structure
```

Step 2: Generate the MODEL with the clustering labels

```
[model,data.y] = cmeans( data.X, 5);           %Number of centers=5
```

Step 3: Display the obtained model and the Voronoi regions.

```
title('The GLA for K=5');
xlabel('X1');
ylabel('X2');
ppatterns(data);                               %Plot the data
ppatterns(model,12);                           %Plot the model Centers
pboundary( model );                            %Plot the Voronoi Region
```



Output Analysis

Type ‘model’ in the command prompt. The output will be somewhat like:

```
model =
      X: [2x5 double]                          %These are the model centers
      y: [1 2 3 4 5]                           %These are the cluster labels for the centers
      t: 12                                    %This is the total number of iterations
  MsErr: [0.4049 0.2208 0.1773 0.1505 0.1398 0.1354 0.1324 0.1290 0.1259 0.1244 0.1228
0.1226]                                        %Errors at each iteration
```

2. Self Organizing Map (SOM): (COURSE WEBSITE)

This package is used for dimensionality reduction. The package provides a batch implementation of the original algorithm proposed by Kohonen [1982] as well as the Minimum Spanning Tree SOM [Kangas et al., 1990].For details about the algorithms please refer to [1].

USER INTERFACE

This package mainly has two interfaces.

1. **loadsomdata(traindata,testdata)**
2. **[mapout, qtest, topmap,training,test] = som(mapdim, knots, b1, b2, itn)**

1. **loadsomdata(traindata,testdata):**
   In order to use the SOM package it is necessary that the data is loaded in the **somkernel** directory in .dat format. the user has the option to load the data sets manually or use this interface. This is simply a helper interface which can be used in conjunction with the main **som** interface.

   - **INPUT PARAMETERS**
     traindata= This is the input data on which the dimensionality reduction is to be done.
     It is an NxD data array where, N=number of samples
     D=dimension of the input sample space.
     testdata= (Structure is same as traindata).This is used to test the model.

   - **TUNABLE PARAMETERS:** None.

- **OUTPUT PARAMETERS:** None.

2. **[mapout,qtest,topmap]=som(mapdim, knots, b1, b2, itn)**
   This is the main interface of the package.The interface is mainly organized as:

   - **INPUT PARAMETERS: None** (By input parameters we identify the parameters such as input samples, input target samples etc.)

   - **TUNABLE PARAMETERS:**
     mapdim: The number of Topological dimensions to be used for the Mapping. For MST specify mapdim=0.

     knots: This is the number of units to be used for each dimensions of the map.

     b1 : The initial neighborhood width to use during training. A Gaussian neighborhood is used. This parameter describes the standard deviation of the Gaussian, where a value of 1 means a standard deviation roughly equal to the width of the map. (The usual value is 1.)
     b2 : The neighborhood width to use at the end of training. Usually a value of .05 to .1 is used, depending on the number of units in the map.

     itn : The number of passes through the training set. Usually 10-50 passes is sufficient.

   - **OUTPUT PARAMETERS:**

     mapout: After program execution, this array will contain the location of each of the units in the sample space.

     qtest: After program execution, this array will contain a quantized version of the test data. For each test sample, the nearest unit will be found, and its location (in the sample space) will be given.

     topmap: After program execution, this array will contain the test file transformed into the topological coordinates. For each test sample, the nearest unit will be found, and its topological coordinate will be given.

EXAMPLE (SOM PACKAGE)

Here we illustrate the usage of the package using the “doughnut” distribution [1]. (see pg 217)

Step 1: Generate the data

```
z=sort(10*rand(50,1));
noise=normrnd(0,0.3,50,2);traindata=[cos(2*pi*z) sin(2*pi*z)]+noise;
z=sort(10*rand(5,1));
noise=normrnd(0,0.3,5,2);testdata=[cos(2*pi*z) sin(2*pi*z)]+noise;
```

Step 2: Load the data

```
loadsomdata(traindata,testdata);
```

Step 3: Generate the SOM mapping on the data provided

```
[mapout, qtest, topmap] = som(1, 5,1, 0.05, 20);
```

Here we select,

Output map dimension =1
Number of Knots per dimension=5
The initial neighborhood width =1;
Final Neighborhood width=0.05
The number of passes through the training set =20

Step 4: We get the output as:

```
mapout =

   0.1414   -0.8903
  -0.7809   -0.6531
  -0.8713    0.7351
   0.2435    1.1763
   0.7636    0.3978
```

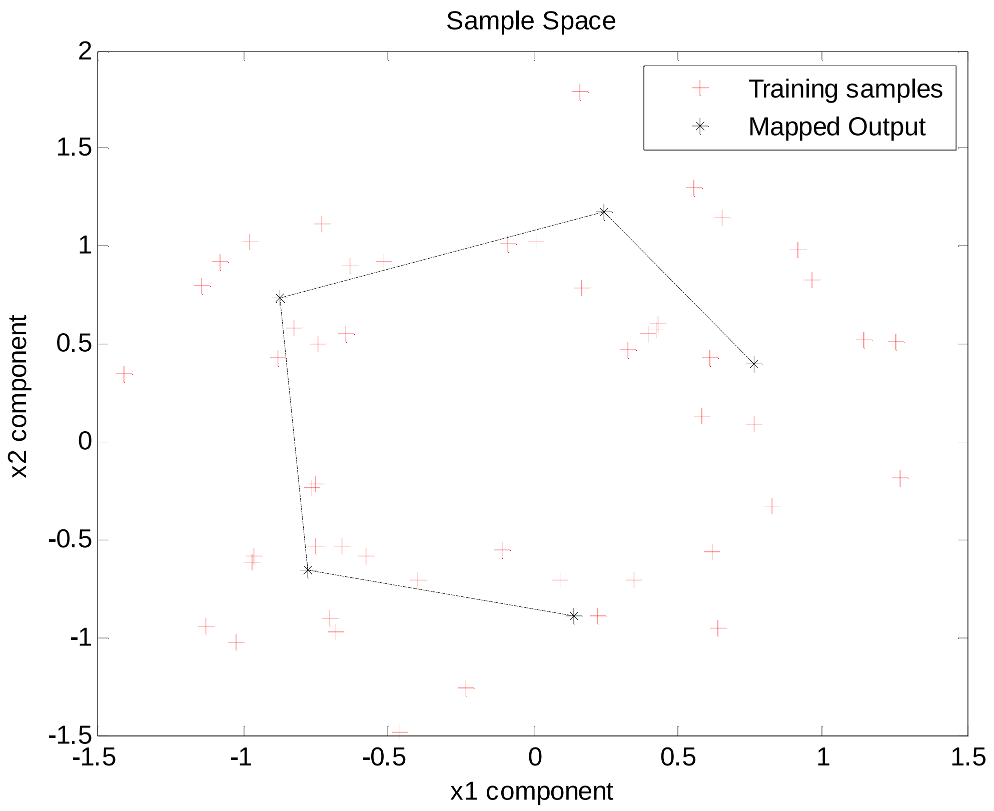
```
quantized version of test data
qtest =


   -0.8713    0.7351
   -0.7809   -0.6531
   -0.7809   -0.6531
    0.1414   -0.8903
    0.1414   -0.8903

topological mapping of test data
topmap =
   3
   2
   2
   1
   1
```

Note: As a special case we can get results for the Minimum spanning Tree (MST) by setting mapdim=0

<u>HOW TO INTERPRET THE RESULTS GRAPHICALLY</u>

The way that we can interpret the output is that in the sample space the output can only take the values that are given in mapout array. Observe that all the qtest (quantized Test values) take the values from the set described by the mapout array. We can also view the results graphically by using the following tactics. (Only for 2D sample space representation)

```
h1=plot(traindata(:,1),traindata(:,2),'r+');           %Plot the data in Sample space
xlabel('x1 component');ylabel('x2 component');
title('Sample Space');
hold on;
h2=plot(mapout(:,1),mapout(:,2),'k*');                 %Plot the Mapped o/p in Sample space
h3=plot(mapout(:,1),mapout(:,2),'k--');                %Connect the Mapped o/p points
legend([h1,h2],'Training samples','Mapped Output');
hold off;
```


Sample Space
Training samples
Mapped Output
x1 component
x2 component
-1.5
-1
-0.5
0
0.5
1
1.5
2

CLASSIFICATION

The Classification package used for the course is mainly STPRTool.

In this report we provide some of the commonly used routines for classification.viz,

1. K Nearest Neighbor Classification**.(KNN)[**Non Parametric]
2. Fisher's Linear Discriminant Analysis**.(FLD**)[Parametric]
3. Quadratic Decision boundary.[Parametric]
4. Support Vector Classifiers.[Parametric]

In addition we shall also provide an additional reference for the CART (Decision Trees).For details of the above methods please refer to [1].
NOTE: The CART toolbox is provided by MATLAB and not the STPRTool.

For the present case we shall also consider the CTM Classifier provided in the course website. However we intend to notify that the interface is likely to change soon.

1. K NEAREST NEIGHBOR CLASSIFIER

USER INTERFACE

The main interfaces provided for this method are:-
1. model=**knnrule**(data,K)
2. y = **knnclass**(X,model)

1. **model=knnrule(data,K):** This interface is used to estimate the model. The briefs of the interface are provided below.

   - **INPUT:**
     data: This is a data structure contains the input training data. The data structure has mainly 4 parameters.

     data.X= a DxN array of input independent variables.
     Here,
     D=Dimension of the input data.
     N=Number of Samples.
     data.y= an 1xN array of Class labels**.**
     data.num_data= Total number of samples
     data.dim= Input data Dimension

- **TUNABLE PARAMETER:**
  K= the number of nearest neighbors.

- **OUTPUT:**
  model= A data Structure which contains the model characteristics.(Note KNN is a non parametric classification. So output model parameters are provided)
  The **model** is the same as the **data** except it has two more attributes added to the structure:
  model. fun=knnclass
  model. K= the number of nearest neighbors (provided by the user).

(Note: Such a modeling may seem inappropriate for the time being. But we shall justify this in the examples that we shall provide later)

2. y = **knnclass**(X,model):This interface is used to classify new data .

- **INPUT:**
  X=The X values of the test data to be classified
  model= The model that was determined using the knnrule on the training data.
- **TUNABLE PARAMETERS:** None
- **OUTPUT:**
  y= the Predicted Class labels for the test data provided**.**

EXAMPLE (Basic Usage)

Step 1:

Load the training data:

```
trn = load('riply_trn');            %riply_trn is available in the STPRTool dataset
```

The Structure of the trn data is:

```
display(trn);
        trn =
        X: [2x250 double]              %The independent training variables
         name: 'Finite data set'       %This is optional
         y: [1x250 double]             %The Class Labels for the 250 samples
        dim: 2                         %The dimension of Input data
        num_data: 250                  %Total number of samples
```

Step 2:

Estimate the Model.(using the training samples)

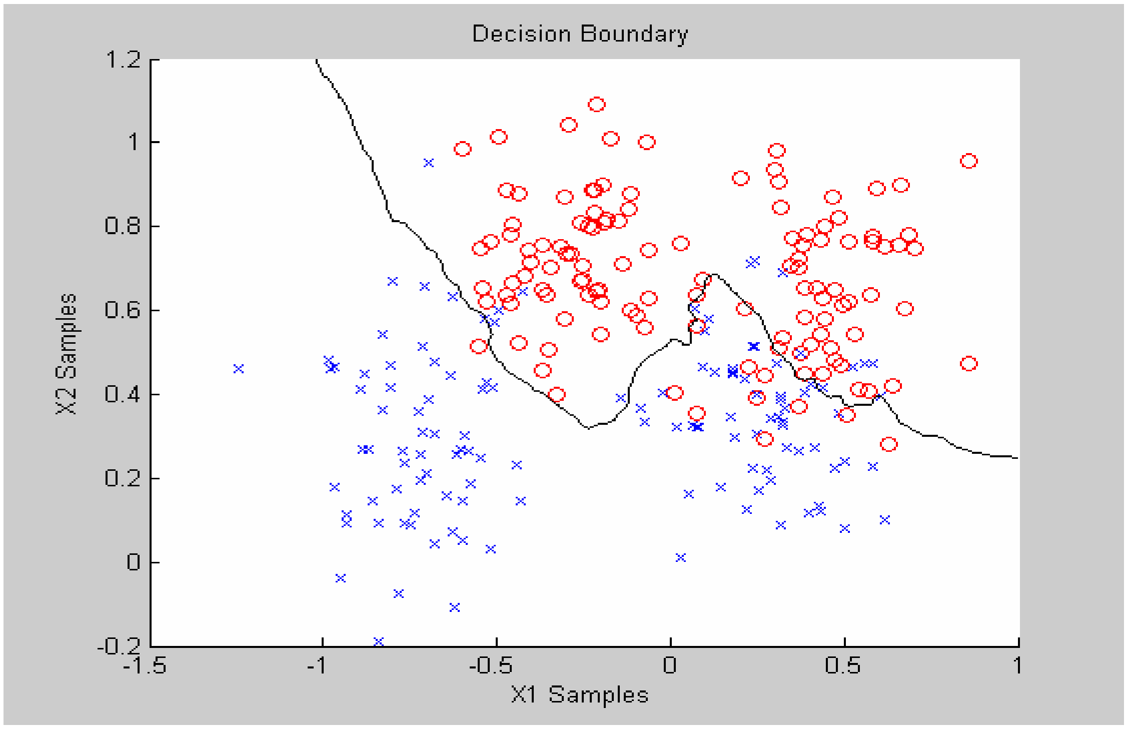
```
model=knnrule(trn,20);              % create model with training data
```

Step 3: Display the Model Generated:

```
figure;
ppatterns(trn);                              %Produces the Scatter Plot of the data
pboundary(model);                   %Displays the Decision boundary
title('Decision Boundary'),xlabel('X1 Samples'),ylabel('X2 Samples');
```

Step 4: Test the Model on test data provided:

```
tst = load('riply_tst');            %Here we generate the Test data
ypred = knnclass(tst.X,model);      %Classify the test data using the model.
err=cerror( ypred, tst.y );         %Calculate the Error on the test dataset.
```



SOME INSIGHTS

```
data=load('riply_trn');
data1=data;                         %Use data1 in place of model
data1.fun='knnclass';                        %Just add the .fun parameter
data1.K=20;                                  %Just add the .K parameter
figure; ppatterns(data); pboundary(data1);
```

The output shall be the same as above. The point that we intend to make here is that the model that is returned by the **knnrule** interface is the same as appending the classifier name and the value of K. The main calculation of forming the decision boundary is done by the pboundary interface. (Sometimes small things can do huge tasks!).

Moreover we justify that still this is not inappropriate. We identify that the KNN Rule is a nonparametric method. So there cannot be a parametric representation of the method. The KNN rule can be seen more as a concept. A better analogy can be that between a class and an object in object oriented programming. Class is simply a prototype and has no entity and the object has some form of entity and can be represented.

2. FISCHER'S LINEAR DISCRIMINANT CLASSIFIER

USER INTERFACE

The main interfaces for this method are:-

- model = **fld**(data)
- [ypred,dfce]= **linclass**(X, model)

1. model = **fld**(data): This interface is used to estimate the model. The briefs of the interface are provided below.
    - **INPUT:**
      data: This is a data structure contains the input data. The data structure has mainly 4 parameters.

      data.X= a DxN array of input independent variables.
      Here,
      D=Dimension of the input data.
      N=Number of Samples.

      data.y= a 1xN array of Class labels.

      data.num_data= Total number of samples
      data.dim= Input data Dimension

    - **TUNABLE PARAMETER:**
      None

    - **OUTPUT:**
      model= A data Structure which contains the model parameters.

      model.W= The coefficients of the Linear Classifier. For binary classification it will be an [Dx1] array.
      Here,
      D=Input Dimension.

      model.b= The Bias term for the classifier. For binary classification will be a 1x1 array i.e. a scalar value.

2. ypred = **linclass**(X,model): This interface is used to classify new data and to test the performance of the model on some test data.

   - **INPUT:**
     X=The X values of the test data. This is an array of the same dimension as the data.X input to the linear classifier used (fld interface above)
     model= The model that was determined using a linear rule (e.g fld or perceptron etc.) on the training data.

   - **TUNABLE PARAMETERS:**
     None
   - **OUTPUT:**
     ypred= the Predicted Class labels for the test dataset provided.
     dfce= The absolute values of the discriminant function before applying the discriminant rule

Note: The model being parametric has a number of parameters. This may however encourage the user to tune the parameter values to their own use. However we point out that the interface of interest here is mainly the FLD to develop the Fischer's Linear Discriminant Classifier. The functionality of the linclass interface will not be explored in the present document.

EXAMPLE (BASIC USAGE)
Step 1

Load the training data:

```
trn = load('riply_trn');                %riply_trn is available in the STPRTool dataset
```

Note the structure of the trn data:

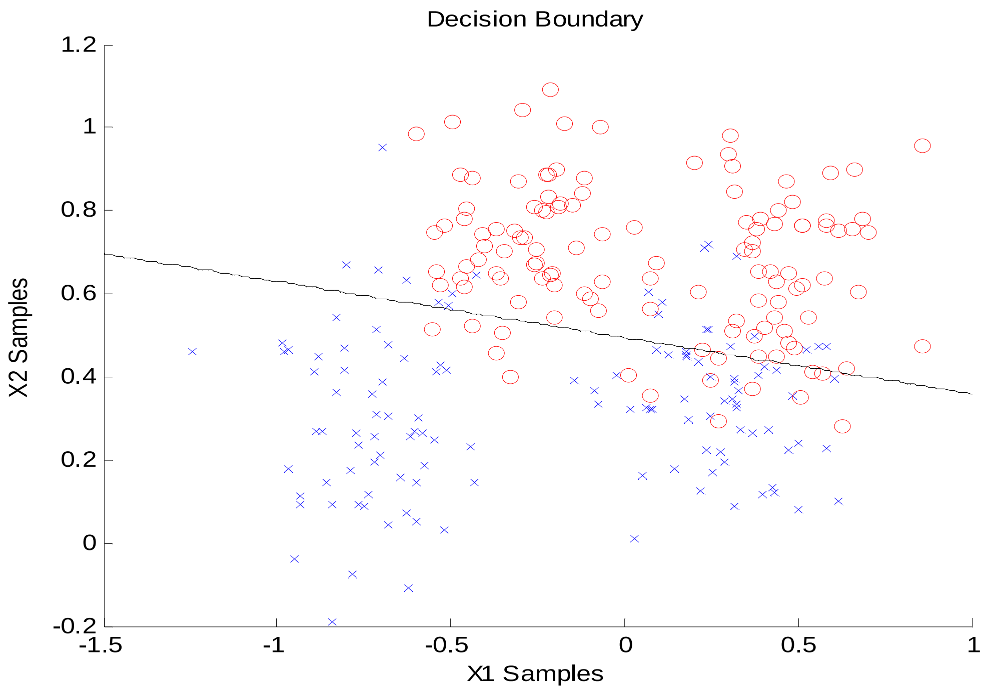
```
display(trn);
        trn =
         X: [2x250 double]               %The independent training variables
         name: 'Finite data set'         %This is optional
         y: [1x250 double]               %The Class Labels for the 250 samples
        dim: 2                           %The dimension of Input data
        num_data: 250                    %Total number of samples
```

Step 2:Estimate the Model.(using the training samples)

```
model=fld(trn); % create model with training data
```

Step 3: Display the Model Generated:

```
figure;
ppatterns(trn);                    %Produces the Scatter Plot of the data
pboundary(model);                  %Displays the Decision boundary
title('Decision Boundary')
,xlabel('X1 Samples'),ylabel('X2 Samples');
```



Step 4: Test the Model on test data provided:

```
tst = load('riply_tst');                  %Here we generate the Test data
ypred = linclass(tst.X,model);            %Classify the test data using the
err=cerror( ypred, tst.y );          %Calculate the Error on the test dataset.
```

3. QUADRATIC DECISION BOUNDARY

Here we provide two different concepts of estimating the quadratic decision boundary:

METHOD 1
1. Map the input samples to a feature space by using quadratic mapping and perform a linear decision rule in the feature space. Now a linear model in the mapped space will be equivalent to a quadratic model in the input sample space.

METHOD 2
2. Use a quadratic discriminant function instead.

<u>**Using METHOD 1**</u>

USER INTERFACE
In this case the SPRTool provides mainly 2 interfaces to apply this method:

- outdata = **qmap**(data)
- quad_model = **lin2quad**(lin_model)

1. outdata=**qmap**(data): The utility of this interface revolves around the fact that we need to map the input sample space using a quadratic mapping. Thus qmap will map input n-dimensional data X into a new (D*(D+3)/2)-dimensional space using the quadratic mapping. D=Input Dimension

- **INPUT**

data = This is a data structure containing the input data. The data structure has mainly 4 parameters.

data.X= a DxN array of input independent variables.
Here,
D=Dimension of the input data.
N=Number of Samples.

data.y= an 1xN array of Class labels.

data.num_data= Total number of samples
data.dim= Input data Dimension

- **TUNABLE PARAMETERS**:
  None

- **OUTPUT**:
  outdata: The output mapped data.

  outdata.X= The newly mapped features in the feature space. This is a (D*(D+3)/2)xN array.

  Here,
  D=Dimension of the input data.
  N=Number of Samples.

  outdata.y=same as data.y(as above)

2. quad_model = **lin2quad**(lin_model)

The utility of this tool revolves around the fact that the linear model obtained by estimating a linear decision boundary in the quadratic mapped feature space has to be converted to a quadratic model in the sample space.

- **INPUT**
  lin_model: This is a linear model obtained by using a linear model in the feature space. Eg: Fischer's LDA or Perceptron etc. The model should mainly have 2 parameters:

  lin_model.W=Array of coefficients/Parameters of the Linear Model.
  lin_model.b=Bias term for the Linear model.

- **TUNABLE PARAMETERS:** None

- **OUTPUT**
  quad_model: The quadratic model. It mainly has three parameters.

  quad_model.A=Hessian of the Discriminant function
  quad_model.B=The Coefficients of the Linear Terms of the discriminant function.
  quad_model.C=The constant term for the discriminant function.

Using Method 2

The SPRTool also provides two main interfaces

- model = **mlcgmm**(data,cov_type)
- quad_model = **bayesdf**(model)

1.model = **mlcgmm**(data,cov_type): This is the interface provided to compute the Maximum Likelihood estimation of parameters of Gaussian mixture model for given labeled data sample.

- **INPUT**
  data = This is a data structure containing the input data. The data structure has mainly 4 parameters.

  data.X= a DxN array of input independent variables.
  Here,
  D=Dimension of the input data.
  N=Number of Samples.

  data.y= an 1xN array of Class labels.

  data.num_data= Total number of samples
  data.dim= Input data Dimension

  cov_type= specifies the shape of covariance matrix
  cov_type = 'full' full covariance matrix (default)
  cov_type = 'diag' diagonal covariance matrix
  cov_type = 'spherical' spherical covariance matrix

- **TUNABLE PARAMETERS**: None (Note that we do not identify the cov_type as a tunable parameter.By tunable parameter we identify the parameters needed to tune the original model. Here the model in consideration is the quadratic model.

- **OUTPUT**
  model : is a data structure which provides the Estimated Gaussian mixture model. It has mainly 3 parameters:
  model .Mean =Mean vectors.
  model.Cov =Covariance matrices.
  model.Prior= Estimated a priory probabilities.

2. quad_model = **bayesdf**(model):

This interface is used to provide the quadratic decision boundary for the model. It implements a quadratic discriminant function given as:

```
f(x) = x'*A*x + B'*x + C
   where, the classification strategy is
      q(x) = 1  if f(x) >= 0,
           = 2  if f(x) < 0.
```

- **INPUT**
  model: The input model is considered to be two multivariate gaussians. This is a data structure with the following parameters.

  model. Mean =Mean values. Is an array of dimension [dim x 2]
  model.Cov=Covariance. Is an array of dimension [dim x dim x 2]
  model .Prior =A priory probabilities. Is an array of dimension [1x 2]

- **TUNABLE PARAMETERS**
  None

- **OUTPUT**

quad_model: Is the output quadratic model. This is a structure with the following parameters.

quad_model.A Quadratic term(Hessian) of the discriminant function. It is an array of dimension [DxD]
D=input dimension
quad_model.B = The Linear term of the discriminant function. It is an array of dimension [D x 1]
quad_model.C = The Bias of the discriminant function. It is a scalar term.

For more details about the concepts of the quadratic decision boundary please refer to [1].For more details about the usage of the tools. Please refer to [2].

TEST INTERFACE:

This interface is used to predict the class labels of the new test data.

USER INTERFACE

ypred = quadclass(tst,model)

This interface is used to classify the input test data using quadratic discriminant function obtained by any of the Methods used earlier. It mainly applied the discriminant rule as:

$$\text{ypred(i)} = \underset{y}{\text{argmax}}\ \text{tst(:,i)'*[model.A(:,:,y)]*tst(:,i) + tst(:,i)'*[model.B(:,y)] +model.C(y)}$$

where, parameters A, B and C are given in model.

- **INPUT**:

tst= Input test data. This is a data structure with the parameters

tst.X=This is the input independent variables of the test data. This is a [Dx N] array with

D=Input dimension

N=Total Number of Input test Samples.

tst.y= Class Labels for the Input Samples. This is a [D x 1] array of class Labels

tst.dim= Dimension of the Input samples. This is a scalar.

tst.num_data= Total Number of samples. This is a scalar.

model= quadratic model determined by any of the above mentioned methods. This is a data structure with the following parameters:

model.A =Quadratic term (Hessian) of the discriminant function. It is an array of dimension [D x D]

D=input dimension

model.B = The Linear term of the discriminant function. It is an array of dimension [D x 1]

model.C = The Bias of the discriminant function. It is a scalar term.

- **TUNABLE PARAMETERS**: NONE

- **OUTPUT**:

  ypred: The Predicted Class Labels. This is a [1 x N] array of class Labels.

EXAMPLE for METHOD 1(Quadratic decision boundary)

Step 1: Input the Training data

```
trn = load('riply_trn');       % load training data
```

Check that the structure of the trn data is same as specified above

```
display(trn);

trn =

      X: [2x250 double]
   name: 'Finite data set'
      y: [1x250 double]
    dim: 2
num_data: 250
```

Step 2: Map the Input data to a (Quadratic) Feature space.

```
map_data = qmap(trn); % give out mapped data
```

Step 3: Perform Linear Discriminant Analysis in the Quadratic Space

```
lin_model=fld(map_data);
```

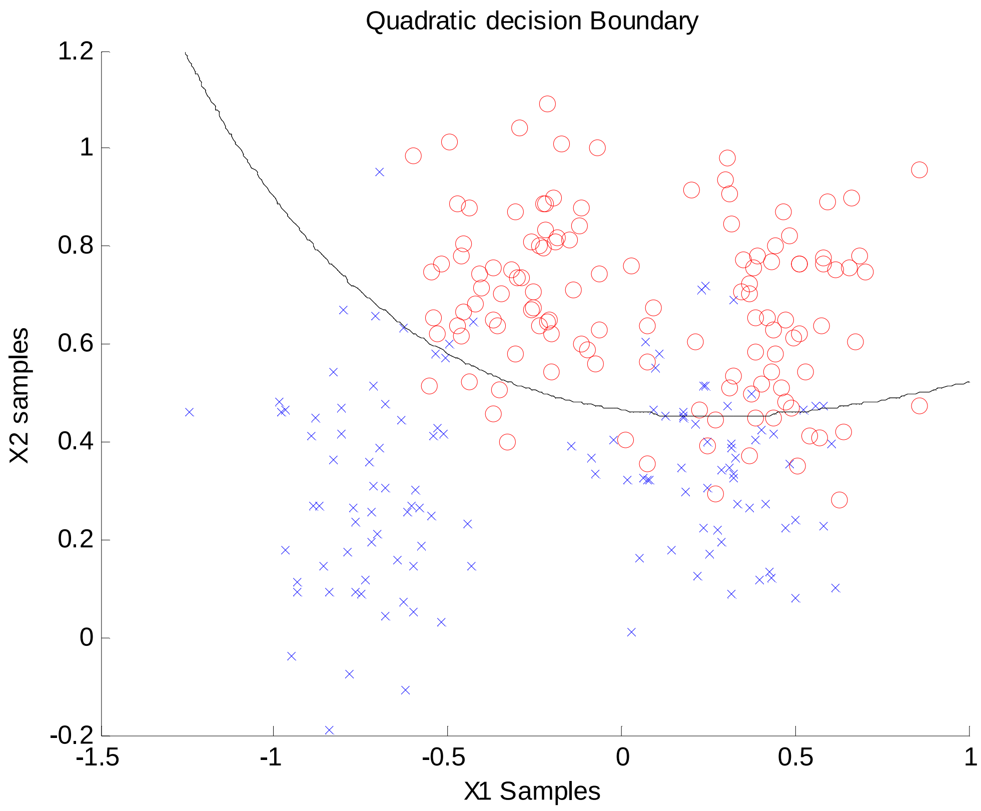
Note: The choice of linear model here is totally on user's discretion. The user may feel it appropriate to use the Linear Perceptron provided by the SPRTool. However in view of the slow convergence rate of the model for ill Conditioned data we prefer using the LDA. However the user may like to overcome this issue by setting the maximum number of iteration to some lower value to ensure faster results.(With an obvious tradeoff with quality).

Step 4: Change the Linear Model to a Quadratic Model(In sample space)

```
quad_model = lin2quad(lin_model);      % change linear model to quadratic model
```

Step 5: Display the Decision Boundary

```
figure; ppatterns(trn);                % plot original data and show quadratic model
pboundary(quad_model);
xlabel('X1 Samples'),ylabel('X2 samples'),title('Quadratic decision Boundary');
```



Step 6: Test the Model on a Test data

```
tst=load('riply_tst');                              %Load t he Test data
ypred = quadclass(tst.X, quad_model);                      %Determine the Predicted Class Labels
err=cerror(ypred, tst.y);                                  % Find  Classification Error
display(sprintf('The Classification Error for the model is %f',err));
```

EXAMPLES for METHOD 2(Quadratic decision boundary)

Step 1: Input the Training data

```
trn = load('riply_trn');       % load training data
```

Check that the structure of the trn data is same as specified above

```
display(trn);
trn =
          X: [2x250 double]
       name: 'Finite data set'
          y: [1x250 double]
        dim: 2
   num_data: 250
```

Step 2: Construct decision boundary with a Quadratic Discriminant Function.

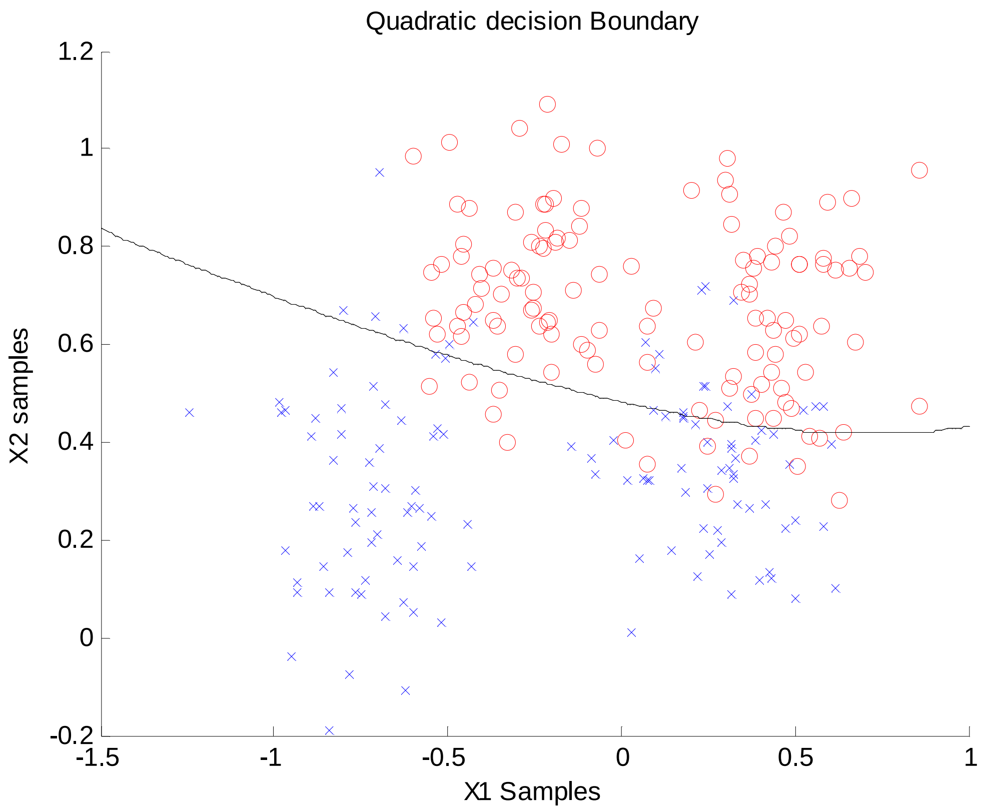
```
gauss_model = mlcgmm(trn);            % create Gaussian mixture model
quad_model = bayesdf(gauss_model);          % create quadratic discriminant model
```

Step 3: Display the Decision Boundary

```
figure; ppatterns(trn);                  % plot original data and show quadratic model
pboundary(quad_model);
xlabel('X1 Samples'),ylabel('X2 samples'),title('Quadratic decision Boundary');
```

Step 4: Test the Model on a Test data

```
tst=load('riply_tst');                                  %Load t he Test data
ypred = quadclass(tst.X, quad_model);                        %Determine the Predicted Class Labels
err=cerror(ypred, tst.y);                                    % Find  Classification Error
display(sprintf('The Classification Error for the model is %f',err));
```



4. SUPPORT VECTOR CLASSIFIERS

The support vector machine (SVM) is a universal constructive learning procedure based on the statistical learning theory (Vapnik 1995). For details about the support vector classifiers (SVC) kindly refer to [1].

The STPRTool provides a number of interfaces for SVC. However in context of the Course requirement we identify 3 interfaces for the implementation of Support Vector Classifiers.

- [model,Errors] = **evalsvm**(trn_data,val_data,options)
- [ypred,dfce] = **svmclass**(X,model);
- h=**psvm**(model)

1. **[model,Errors] = evalsvm(trn_data,val_data,options):** This interface is used to **train** and **evaluate** the Support Vector Machines Classifier.

    - **INPUT**

      trn_data : This data structure contains the input training data. The data structure has mainly 4 parameters.

      trn_data.X= a DxN array of input independent variables.
      Here,
      D=Dimension of the input data.
      N=Number of Samples.

      trn_data.y= an 1xN array of Class labels.

      trn_data.num_data= Total number of samples
      trn_data.dim= Dimension of Input data

- **PARAMETER TUNING**

  val_data: This data structure contains the input validation data. The parameters of the data structure are same as the trn_data. It is only needed to be specified if the model selection has to be done on a validation data set.

  options: This specifies the set of options on which the SVM classifier has to be evaluated. It is a data structure that contains the following parameters:-

  options.ker= Specifies the type of Kernel for the SVM Classifier.The different types of Kernels are:-

'linear' : linear kernel H(x,x') = x^T*x'
'poly' :polynomial H(x,x') = (x^T* x'+arg[2])^arg[1]
'rbf' :RBF (Gaussian) H(x,x') = exp(-0.5*||x-x'||^2/arg[1]^2)
'sigmoid' : Sigmoidal H(x,x') = tanh(arg[1]*(x^T*x')+arg[2])
(Please refer to [1] pg 429 for information about setting the parameters)

options.dimarg=Specifies the dimension of arguments for the kernel type. E.g: For ker='rbf' the dimarg=1 and for ker='sigmoid' the dimarg=2

options.arg=Specifies the set of arguments for the Kernel over which the SVC is to be evaluated. It is generally a vector of dimension [dimargx1].We can however set a range of arguments. In such a case the dimension of this parameter changes to [dimargxN].
Where, N=number of arguments we need to test the model for.

options.C= Specifies the set of regularization constants. (Also called as the constraints) over which the SVC is to be evaluated. We can set a range of C values over which the model needs to be evaluated.

options.solver= Specifies the type of Solver to be used by the SVC. **(**default 'smo'**)**

options.num_folds= If the SVC is to be evaluated via Cross Validation rather than on a Validation set, this parameter specifies the number of folds of Cross-Validation that need to be performed for evaluating the model.(default 5)

options.verb = The progress info is displayed if options.verb is set to 1 (default 0).

- **OUTPUT**

model: This is the best model selected by the evalsvm interface based on the validation set error or the Cross Validation error (whichever was specified).The model is a data structure with the following parameters.

model.Alpha= The optimal Lagrange multipliers obtained by solving the dual problem.
model.b= The bias term in the decision function.
model.nsv=Number of Support vectors

model.trnerr=The error on the training data due to the best model**.**
model.margin=The soft margin. This is used by psvm interface while displaying the soft margin.
model.sv=Is a structure containing all the support vectors.
model.options=The options used by the Solver.
model.fun=The type of classifier to be used while displaying the decision boundary. (Used by the PSVM interface)
model.cputime=Time taken to build the model.

Errors: This is the classification error provided by the best model on the Validation set. This may also represent the Cross Validation error if the Validation set is not provided.

ISSUE: One issue with the evalsvm interface (in general for any SVM solver interface for STPRTool) is that we need to specify some argument for the 'linear' SVM. For the other solver interfaces loke 'smo' this value is taken to be one by default. But in case of the evalsvm interface the user needs to specify it. This somewhat confuses the user because a linear SVM can have no arguments. Although this parameter is not used internally it is suggested to set this value to be 1.

2. **[ypred,dfce] = svmclass(X,model):** This is used to classify new test data based on the SVM Classifier that we obtained. For binary classification the discriminant function is:

   y(i) = 1 if f(X(:,i) >= 0
   = 2 if f(X(:,i) < 0

   where f is the discrimiant function given by Alpha [nsv x 1], b [1x1] and support vectors sv.X.

- **INPUT**:

  X: Input vectors to be classified. It should have the same dimensions as the trn.X used to evaluate the SVM Classifier model.

  model: This is the SVM Classifier model. It mainly has the following parameters.

  model.Alpha =Multipliers associated to support vectors. The dimension is [nsv x nfun]
  model .b=Biases.The dimension is [nfun x 1]

model.sv.X =The X values of the Support vectors. The dimension is [D x nsv]
model.options.ker = The type of Kernel . This is a string.
model.options.arg =The Kernel argument for best model.
where,
nsv= Number of Support Vectors.
nfun= The number of discriminant functions. For Binary case this is 1
D= Input dimension.

- **TUNABLE PARAMETERS:** None
- **OUTPUT:**
  ypred=The predicted labels of the input test data. This is a vector of dimension [1 x N]
  N=Number of samples
  dfce= Values of discriminant functions**.** This is a matrix of dimension [nfun x N].

3. **h=psvm(model):**This interface is used to plot the SVM Decision boundary along with the Soft Margin.
   - **INPUT:**
     model: This is the best model obtained by using the **evalsvm** interface.(see above for the model parameters)
   - **TUNABLE PARAMETERS:** None.
   - **OUTPUT:** The handler to the graphical object.

EXAMPLE (SUPPORT VECTOR CLASSIFIER)
In this example we shall set the range of C values from [1, 10, 20, 30].We shall use Kernel type as ‘RBF’ and set the range of σ = [0.1, 0.5, 1, 5]. The model selection will be done based on 15 fold Cross Validation. (For more information on the SVM Model Selection please refer to [1] pg 446)

STEP 1: Load the Training data

```
trn = load('riply_trn');
```

STEP 2: Define the model parameters(Parameter Tuning)

```
options.ker = 'rbf';                    %Kernel Type is ‘RBF’
options.arg = [0.1 0.5 1 5];            %The model will be evaluated on a range of σ values
options.C=[1,10,20,30];         % Specify the Range of C values
options.solver = 'smo';         %Specify the Type of Solver
options.num_folds = 15;         %Specify the number of Folds for Cross Val
 options.verb = 1;                      %Set this to 1 if you need to print the CV Errors
```

STEP 3: Perform Model selection.

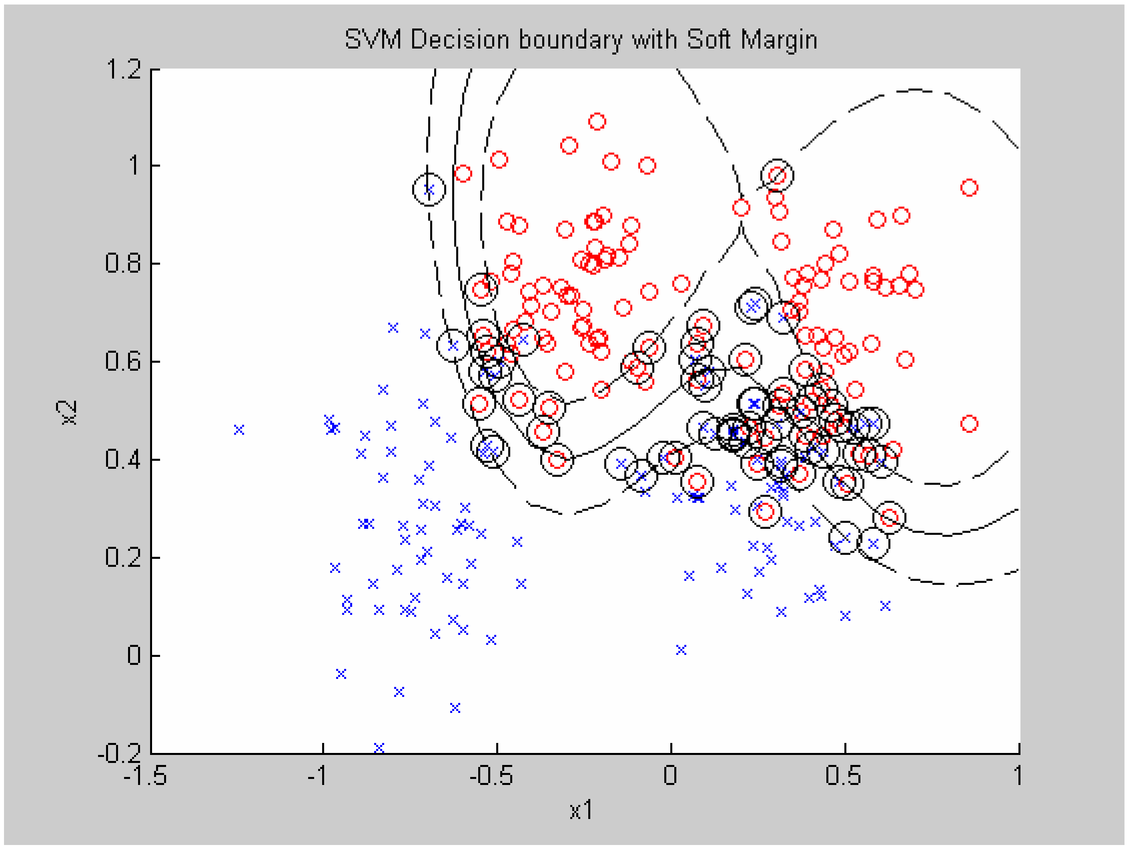
```
 [model,Errors] = evalsvm(trn,options);  %Use the interface for selecting the best Model
```

STEP 4: Now that we have all the CV Errors and the best model let us test the model on some test data. In this case we shall use the Ripley’s test data provided in STPRTool.

```
tst = load('riply_tst');
 [ypred,dfce] = svmclass(tst.X,model);   %Predict the Class label for the test data
cerror(ypred,tst.y)                             %Test the output
```

STEP 5: Display the Decision boundary with the Soft Margin

```
 figure;  ppatterns(trn); psvm(model);
 hold on;
 xlabel('x1'),ylabel('x2');
 title('SVM Decision boundary with Soft Margin');
```


SVM Decision boundary with Soft Margin
x1
x2
-1.5
-1
-0.5
0
0.5
1
-0.2
0
0.2
0.4
0.6
0.8
1
1.2

5. CONSTRAINED TOPOLOGICAL MAPPING (CTM) Classifier

The CTM is a kernel regression method based on a modification of the self-organizing map(SOM).Here we present the modified CTM method to solve classification problems. for details please refer to [1] pg 377.

Note:

There is still no MATLAB interface for the CTM Classifier yet. However we intend to intimate that it is soon to come. For the present case we shall use the UNIX version of the CTM. It can also be run in Windows using Cygwin. http://www.cygwin.com/. Moreover for using the package we need to save the traindata and the testdata in a file format. We shall provide some suggestions to create a file of that format using MATLAB.(For now we do not provide any interface for loading the data as a MATLAB interface will be available soon enough)

USER INTERFACE

- **ctmb.exe:** This is an interactive interface which shall guide through the entire process. We identify mainly 3 passes of model evaluation.
    - **INPUT:**

      traindata: This is an ASCII input files with tabs or spaces between the columns of data file containing the training data. In the example we have provided a way to create such files.

      testdata: It has the same format as the traindata. This is the data on which the model is tested.

    - **PARAMETER TUNING:**

      number of map dimension: This is the number of dimension of the feature space. It depends upon the type of the problem. However, this should always be less than or equal to the dimension of the input space. For using the Minimal Spanning tree(MST) specify this parameter as 0.

      smoothness level: This number controls the smoothness of the fit. A value of 0 causes bctm to find the model which minimizes the error on the training set, so may select a non-smooth model. A value of 9 causes bctm to find the model which minimizes the cross-validation error estimate of training set, so this will be a smoother model. A number between 0 and 9 will minimize a mixture of these two error measures. It does a 10 Fold Cross Validation.

number of units per dimension: This algorithm requires far fewer knots per dimension than the original CTM algorithm, because a piecewise linear approximation is used. This is usually a number between 1 and 100 depending on the map dimensionality. If a value of 0 is entered, the algorithm attempts to find this parameter automatically, which significantly increases the run time.

adaptive scaling: It specifies if the Adaptive scaling has to be done.

- **OUTPUT:**
  The script provides the output statistics providing the training error, Cross Validation Error and the Test Error. The model out puts can be obtained from the following files:
  "**map**" contains the locations of the units. (Output for step 2 pg 377).

  "**coef**" contains the zero-th order as well as first order coefficients for each unit.(Output for Step 3 pg 377).

  "**fit1**" contains the fitted values of the test set.

EXAMPLE (CTM Classifier)

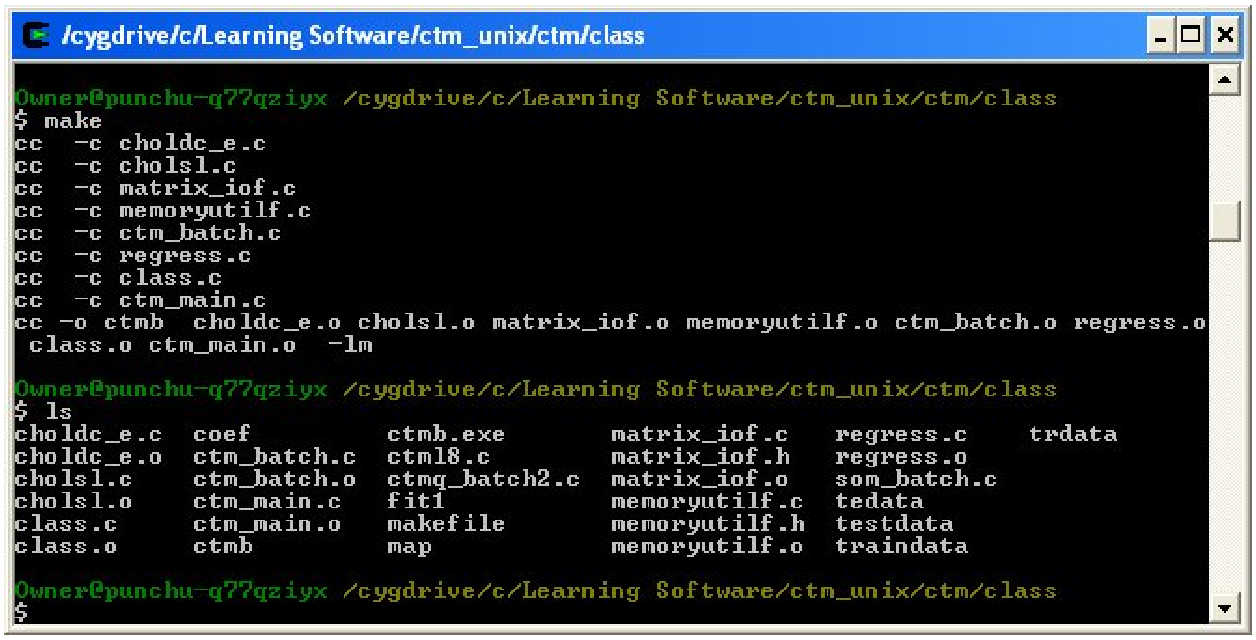
STEP 1: Save the Training and Test data file in the same folder as ctm/**class**
(Note: CTM has two interface regression and class. For classification the interface in class shall be used)

The way to create such file format: (We use the Ripley’s data provided in STPRTool)

```
trn = load('riply_trn');          % load training data
tst = load('riply_tst');          % load testing data
trainX=trn.X';
trainy=trn.y';
trainy(find(trainy==2))=-1;                          %The Class Labels are +1 or -1
train=[trainX,trainy];


testX=tst.X';
testy=tst.y';
testy(find(testy==2))=-1;
test=[testX,testy];
```

save traindata train -ASCII
save testdata test –ASCII

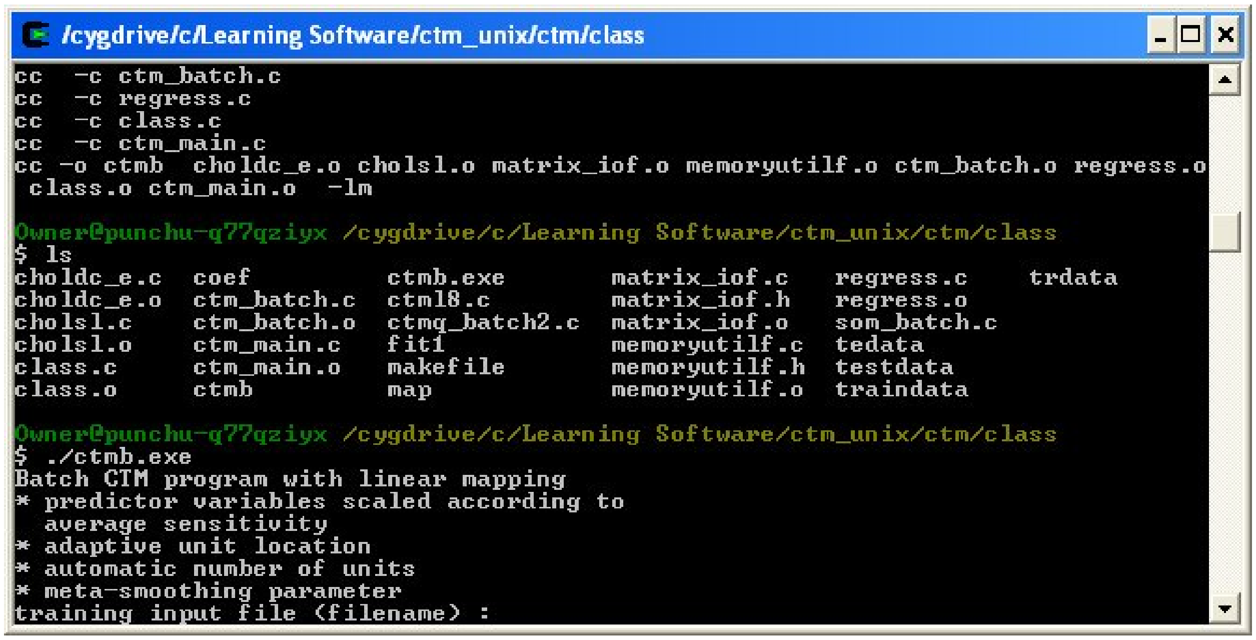
STEP 2: Run the ctmb.exe interface. (Be sure to make the .makefile first).
The rest is interactive



/cygdrive/c/Learning Software/ctm_unix/ctm/class

```
Owner@punchu-q77qziyx /cygdrive/c/Learning Software/ctm_unix/ctm/class
$ make
cc  -c choldc_e.c
cc  -c cholsl.c
cc  -c matrix_iof.c
cc  -c memoryutilf.c
cc  -c ctm_batch.c
cc  -c regress.c
cc  -c class.c
cc  -c ctm_main.c
cc -o ctmb  choldc_e.o cholsl.o matrix_iof.o memoryutilf.o ctm_batch.o regress.o
 class.o ctm_main.o  -lm

Owner@punchu-q77qziyx /cygdrive/c/Learning Software/ctm_unix/ctm/class
$ ls
choldc_e.c  coef          ctmb.exe       matrix_iof.c   regress.c    trdata
choldc_e.o  ctm_batch.c   ctml8.c        matrix_iof.h   regress.o
cholsl.c    ctm_batch.o   ctmq_batch2.c  matrix_iof.o   som_batch.c
cholsl.o    ctm_main.c    fit1           memoryutilf.c  tedata
class.c     ctm_main.o    makefile       memoryutilf.h  testdata
class.o     ctmb          map            memoryutilf.o  traindata

Owner@punchu-q77qziyx /cygdrive/c/Learning Software/ctm_unix/ctm/class
$
```

After the make is complete run the ./ctmb.exe



/cygdrive/c/Learning Software/ctm_unix/ctm/class

```
cc  -c ctm_batch.c
cc  -c regress.c
cc  -c class.c
cc  -c ctm_main.c
cc -o ctmb  choldc_e.o cholsl.o matrix_iof.o memoryutilf.o ctm_batch.o regress.o
 class.o ctm_main.o  -lm

Owner@punchu-q77qziyx /cygdrive/c/Learning Software/ctm_unix/ctm/class
$ ls
choldc_e.c  coef          ctmb.exe       matrix_iof.c   regress.c    trdata
choldc_e.o  ctm_batch.c   ctml8.c        matrix_iof.h   regress.o
cholsl.c    ctm_batch.o   ctmq_batch2.c  matrix_iof.o   som_batch.c
cholsl.o    ctm_main.c    fit1           memoryutilf.c  tedata
class.c     ctm_main.o    makefile       memoryutilf.h  testdata
class.o     ctmb          map            memoryutilf.o  traindata

Owner@punchu-q77qziyx /cygdrive/c/Learning Software/ctm_unix/ctm/class
$ ./ctmb.exe
Batch CTM program with linear mapping
* predictor variables scaled according to
  average sensitivity
* adaptive unit location
* automatic number of units
* meta-smoothing parameter
training input file (filename) :
```

Provide the input file name and the test file name.

STEP 3: Parameter Tuning

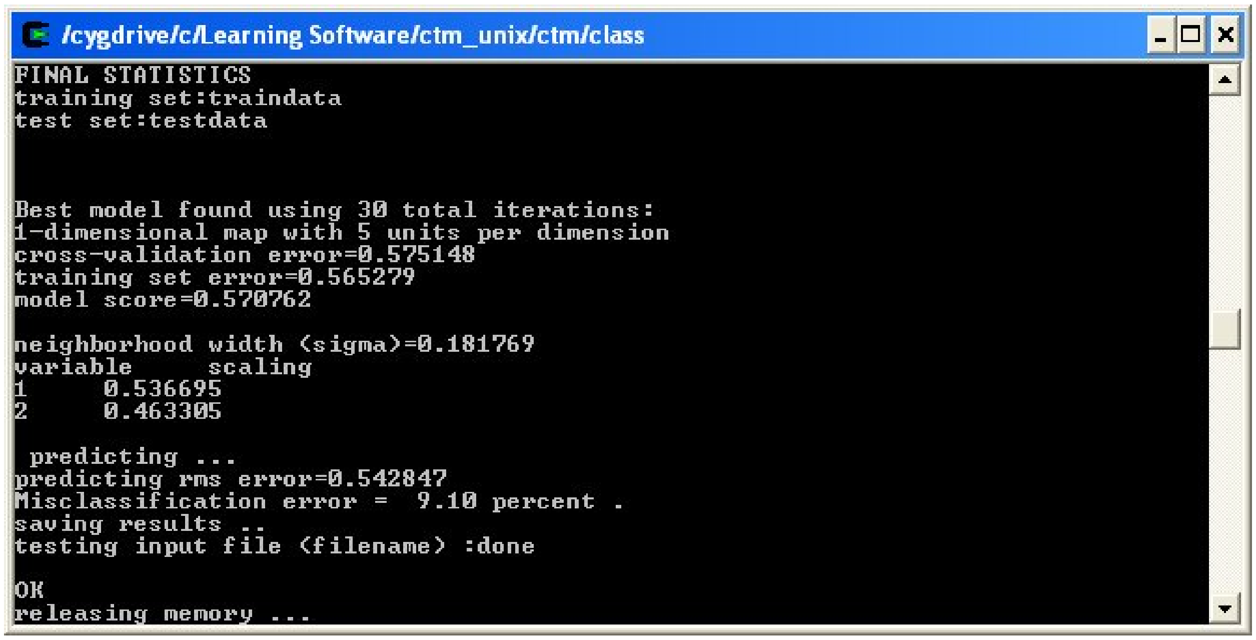
Enter the Map Dimension: 1 (Makes sense to use a lower map dimension than the input dimension)

Enter the smoothness level: 5 (We are unsure of the noise level of the data set. So going for a safer option)

Enter the number of units per dimension: 5
Adaptive Scaling: 1=yes (see pg 377)

Note: The model selection for the CTM Classifier needs further consideration. (Pg 378)

The Output Statistics are provided as:



```
FINAL STATISTICS
training set:traindata
test set:testdata


Best model found using 30 total iterations:
1-dimensional map with 5 units per dimension
cross-validation error=0.575148
training set error=0.565279
model score=0.570762

neighborhood width (sigma)=0.181769
variable      scaling
1      0.536695
2      0.463305

 predicting ...
predicting rms error=0.542847
Misclassification error =  9.10 percent .
saving results ..
testing input file (filename) :done

OK
releasing memory ...
```

Moreover the output files provided are:

map(Consider this as the output of step 2 pg 377)

```
-0.721728     0.290520
-0.464058     0.683904
-0.168106     0.748465
0.450441      0.644347
0.285195      0.326139
```

As we can see it contains the Location of the Units in sample space.

coeff(Consider this as the output of Step 3 pg 377)

w1 w2 b(0th order coeff)

```
1.000001     0.000001    0.000000
-1.870324    -5.205008   -1.677625
0.378717     0.560309    -1.547222
0.564624     -0.803716   -1.362943
1.399982     -0.773489   -1.652593
```

The coefficients of the Linear regression in the y space.

fit: This file contains the fitted value of the data provided in the testdata. This is equivalent to the output if the Discriminant function. We need to apply the discriminant rule:

y(i) = 1 if f(X(:,i) >= 0
= -1 if f(X(:,i) < 0

6. DECISION TREES

TREEFIT fit a tree-based model for classification or regression. It is a mat lab routine. The details of the interface are available in MATLAB Documentation. Here we provide some specifics that will prove helpful for the course work.

We mainly identify 1 interfaces as the Decision tree utility.

- T = **TREEFIT**(X,Y,'PARAM1',val1,'PARAM2',val2,...)

1. **T = TREEFIT(X,Y,'PARAM1',val1,'PARAM2',val2,...):** It creates a decision tree T for predicting response Y as a function of predictors X.
   - **INPUT**

   X= an N-by-M matrix of predictor values.
   Y = either a vector of N response values (for regression) or a character array or cell array of strings containing N class names (for classification).

   - **TUNABLE PARAMETERS**

   For all trees:
   'categorical' Vector of indices of the columns of X that are to be treated as unordered categorical variables
   'method' Either 'classification' (default if Y is text) or
   'regression' (default if Y is numeric)

'names' A cell array of names for the predictor variables, in the order in which they appear in the X matrix
from which the tree was created
'splitmin' A number N such that impure nodes must have N or more
observations to be split (default 10)
'prune' 'on' (default) to compute the full tree and the optimal
sequence of pruned subtrees, or 'off' for the full tree without pruning

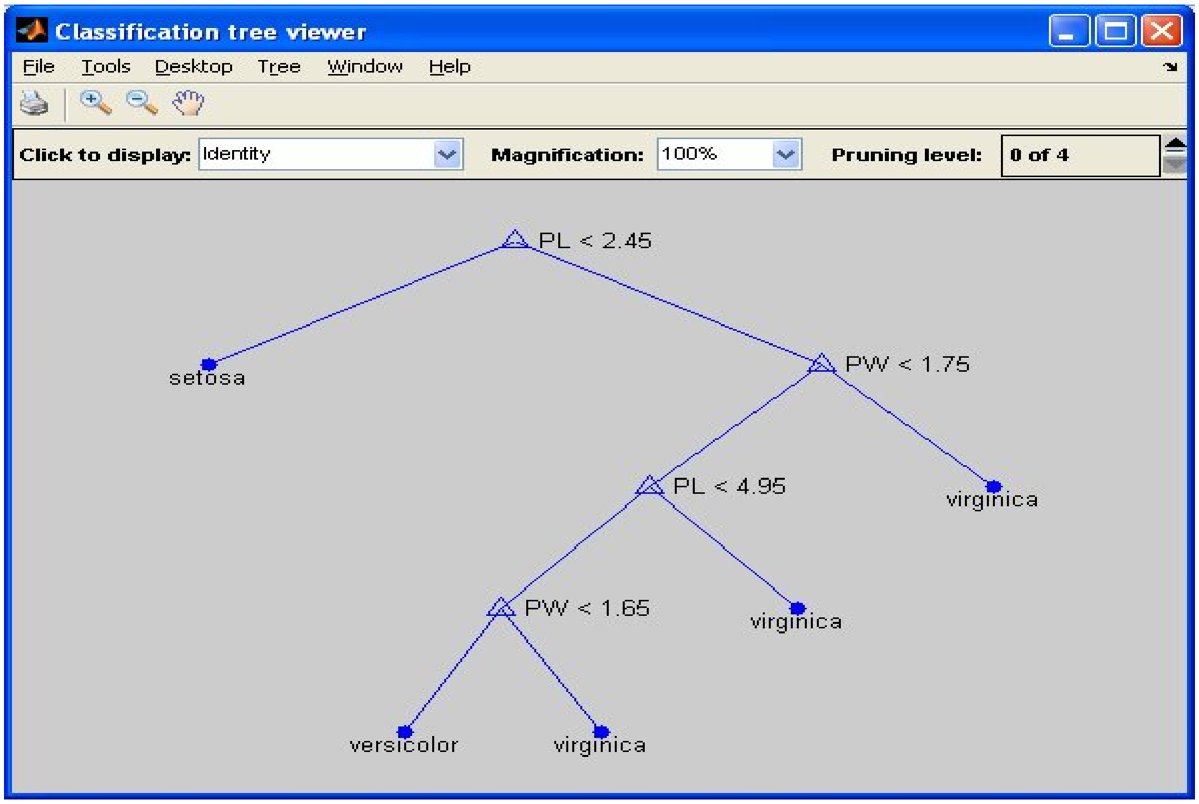
For classification trees only:
'cost' Square matrix C, C(i,j) is the cost of classifying
a point into class j if its true class is i (default
has C(i,j)=1 if i~=j, and C(i,j)=0 if i=j). Alternatively
this value can be a structure S having two fields: S.group
containing the group names as a character array or cell
array of strings, and S.cost containing the cost matrix C.
'splitcriterion' Criterion for choosing a split, either 'gdi' (default)
for Gini's diversity index, 'twoing' for the twoing rule,
or 'deviance' for maximum deviance reduction
'priorprob' Prior probabilities for each class, specified as a
vector (one value for each distinct group name) or as a
structure S with two fields: S.group containing the group
names as a character array or cell array of strings, and
S.prob containing a a vector of corresponding probabilities

OUTPUT
t= a binary tree where each non-terminal node is split based on the values of a column of X. NaN values in X or Y is taken to be missing values, and observations with any missing values are not used in the fit. Mainly it is an object of class classregtree.

To display the Tree we could either use the view(t) or treedisp(t) option.

EXAMPLE for DECISION TREE

```
load fisheriris;                                  % load the Fisher 's  iris data.
 t = treefit(meas, species);                  % create the tree based on the loaded data.
treedisp(t,'names',{'SL' 'SW' 'PL' 'PW'});        % display the tree
```


Classification tree viewer
File Tools Desktop Tree Window Help

Click to display: Identity Magnification: 100% Pruning level: 0 of 4

NOTE: The Help Document Produced for the decision Tree is taken from the MATLAB Help documents.

SOME MORE EXAMPLES USING THE CLASSIFICATION TOOL (STPRTOOL)

Example 1: In this example we shall see the usage of the KNN Classifiers and the SVM Classifiers on the Haberman's survival data set. (available at UCI Database http://archive.ics.uci.edu/ml/datasets/Haberman's+Survival). This data set has 306 training samples (x,y). We shall use only two input variables, Age and Number of Nodes. (Note: The data has been prescaled to the range of [0, 1])

(This is a sample solution to the HW4.Part 2)

Step 1: Process the data

```
load haberman.data -ASCII

X=haberman(:,[1,3]);
y=haberman(:,4);
X(:,1)=scale(X(:,1),0,1);                 %Scale the data to the range of [0,1]
X(:,2)=scale(X(:,2),0,1);                 %Scale the data to the range of [0,1]

trn.X=X';                                 %Preprocess the data
trn.y=y';
trn.dim=2;
trn.num_samp=306;
trn.name='Habermans Survival data';
```

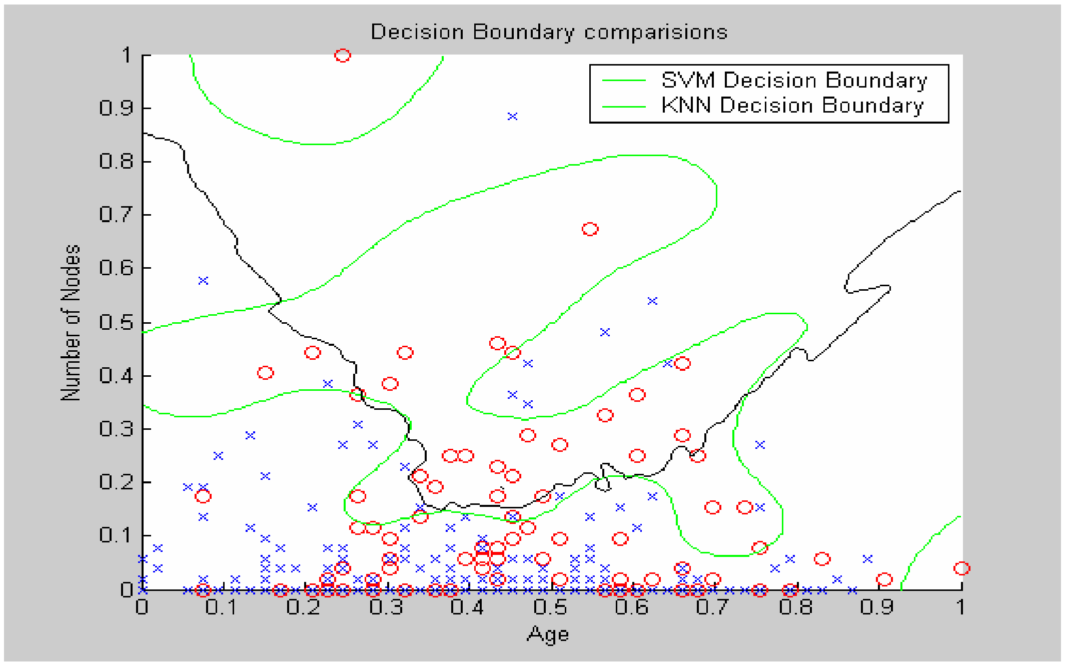
Step 2: Obtain the best SVM Model based on 15-Fold CV error.

```
diary on;                                           % We diary all the output
  C=[1,10,20,30];                                   %Select the values of  C
  display('RBF Kernel SVM using the Exhaustive strategy');
  options.ker = 'rbf';                              %Specify the Type of Kernel
  options.C = C;                                    %Set the Values of C
  options.solver = 'smo';                           %Set the type of solver
  options.verb = 1;                         %Print the CV Error in the screen
  options.arg=[0.1,0.5,1,5];                %Select the range of sigma values
  options.dimarg=1;                         %We may skip this
  options.num_folds = 15;                   %Perform 15 Fold Cross Validation
  [model,Errors] = evalsvm(trn,options);            %Obtain the best model
  figure; ppatterns(trn);psvm(model);                            %Display the best model.
  xlabel('Age'); ylabel('Number of Nodes');
  title('SVM Decision Boundary with Margin');
  model_RBF=model;
diary off;
```

Step 3: Obtain the best model for KNN with K ranging from 1 to 100.

```
%This part is for KNN

    maxK=100;
    traindata=trn.X';
    traindata(:,3)=trn.y';

    for k=1:maxK                                    %Select the maximum value of K
      classerror=0;

     ypred=[];
      for i=1:size(traindata,1)
            [trset,vset] =leave1out(i,traindata);    %Perform Leave 1 Out CV
            trset=trset';
            vset=vset';
            trstrct=struct('X',trset(1:2,:),'y',trset(3,:),'name',...,
                          'Training Data','dim',2,'num_data',size(trset,2));
            model=knnrule(trstrct,k);
            valstrct=struct('X',vset(1:2,:),'y',vset(3,:),'name',...,
                         'Validation Data','dim',2,'num_data',size(vset,2));
            ypred=[ypred;knnclass(valstrct.X,model)];     %Test the model on ValidationSet

      end
      classerror=cerror(ypred,traindata(:,3));
      ModelErr(k,1)=classerror;
      ModelErr(k,2)=k;
    end

    Modelmin=min(ModelErr(:,1));                 %Select the best model with least Validation err
    for i=1:size(ModelErr,1)
      index=find(ModelErr(:,1)==Modelmin);
      optimumK=ModelErr(max(index),2);
    end

    traindata=traindata';
    trainset=struct('X',traindata(1:2,:),'y',traindata(3,:),'name',...,
      'Training Data','dim',2,'num_data',size(traindata,2));

    model_knn=knnrule(trainset,optimumK);      %Obtain the best KNN Model
```

Step 4: Finally we present the two decision boundary obtained.



NOTE: **The main reason behind presenting the MATLAB coding is to enumerate the utility of the evalsvm interface. We observe that using that interface we can reduce the number of lines of code to a significant degree. We can perform all the complex computation like: Evaluating on a validation set, Evaluating a model by Cross Validation etc using minimal lines of coding. This aptly reflects the ease of using the evalsvm interface.**

EXAMPLE 2

In this example we simply try to explore the SVM Classifier a bit more. We consider the WISCONSIN BREAST CANCER data set. The data set has Number of instances: 569 and Number of attributes: 32 (ID, diagnosis, 30 real-valued input features).This is publicly available at http://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+%28Diagnostic%29

For the present case we shall use the SVM Classifier of 4 types of kernels:

1. Linear.
2. Polynomial of degree [2,3,4]
3. RBF with sigma[0.1,0.5,1]
4. Sigmoid with γ=2 and a=1.

The range of C value is [0.1,1,10,50]

Moreover we perform the model selection based on a 10 Fold Cross Validation error.

```
C=[0.1,1,10,50];    %Set the range of C values

%***************LINEAR SVM********************
    display('Linear SVM using the Exhaustive strategy');
    options.ker = 'linear';
    options.C = C;
    options.solver = 'smo';
   options.verb = 1;
   options.arg=1;                                   %Check that the argument is set to 1
   options.num_folds = 10;
   [model,Errors] = evalsvm(data,options);
   model_linear=model;

%****************POLYNOMIAL SVM*****************
   display('Polynomial Kernel SVM using the Exhaustive strategy');
   options.ker = 'poly';
  options.C = C;
 options.solver = 'smo';
   options.verb = 1;
   options.arg=[2,3,4];
   options.dimarg=1;
   options.num_folds = 10;
   [model,Errors] = evalsvm(data,options);
   model_poly=model;
```

```
%*************RBF SVM***********************
display('RBF Kernel SVM using the Exhaustive strategy');
   options.ker = 'rbf';
  options.C = C;
    options.solver = 'smo';
   options.verb = 1;
   options.arg=[0.1,0.5,1];
   options.dimarg=1;
   options.num_folds = 10;
   [model,Errors] = evalsvm(data,options);
   model_RBF=model;

%************SIGMOIDAL SVM********************
   display('Sigmoid Kernel SVM using the Exhaustive strategy');
   options.ker = 'sigmoid';
    options.C = C;
    options.solver = 'smo';
   options.verb = 1;
   options.arg=[[2;1]];        %The arguments are selected in conformance with MERCER rule
   options.dimarg=2;
   options.num_folds = 10;
   [model,Errors] = evalsvm(data,options);
   model_Sig=model;
```

OUTPUT

The Cross Validation errors for the different methods for their best Models are tabulated below.

Method	**Value of C**	**Kernel Parameters**	**Cross Validation Error**
Linear SVM	1	None	0.0228
Polynomial SVM	1	Degree=2	0.0211
SVM(RBF)	1	Sigma=1	0.0211
SVM(Sigmoid)	0.1	γ=2 and a=1.	0.3725

REGRESSION(COURSE WEBSITE)

In this section we mainly identify two types of methods

1. Non Adaptive methods or Linear Estimators.
2. Adaptive Methods
    - Adaptive Dictionary Methods.
    - Adaptive Kernel Methods.

NON ADAPTIVE METHODS(LIN/PLIN Regression)

The course website mainly provides one interface for the non-adaptive methods. This is the **regression** package provided. However for the sake of simplicity we introduce the **regrIntf** interface which internally use this package and at the same time provides better usability. This interface can be used for Linear as well as Penalized Linear estimators. The basis functions used are 'Polynomial' and 'Trigonometric' functions. The interface will provide the best model (optimal complexity) over a Training dataset and output the Mean Squared Error for the selected model over the Test Data. It provides for the user to specify the model selection criteria viz,

Analytic Criteria:

1. Final Prediction Error (FPE).
2. Schwartz Criterion (SC).
3. Generalized cross-validation (GCV).
4. Shibata's model selector (sms)
5. Vapnik's Measure (VM)

Cross-Validation:

1. Leave-one-out Cross Validation.
2. M-Fold Cross Validation.

It also allows the user to specify the maximum degree of the basis function. For a brief description of all the above methods please refer to [1]

USER INTERFACE

The main interface to use the package is:

[model,mse]=**regrIntf**(traindata,testdata,rtype,bfun,msel,degr)

INPUT:

traindata: The traindata is a data structure which has two parameters X and Y.

traindata.X= an nx1 or nx2 array of input data samples. This is mainly the array of the independant variables. Here the total number of samples is n and the dimension of the input variables is 1 or 2. (This package supports for the maximum dimension of the input parameters to be 2. For details of the other limitations of the package please refer to the section 3)

traindata.Y= an nx1 array of target samples. This is the desired target of the unknown system that needs to be determined by the Estimator.

testdata.X= (same as traindata.X).The only difference is that these data samples will be used for testing the model performance.

testdata.Y= (same as traindata.Y).The only difference is that these data samples will be used for testing the model performance.

TUNABLE PARAMETERS

rtype= This parameter specifies the type of regression to be performed. The package supports only two types of regressions:

'LIN' ≡ Linear Regression.

'PLIN' ≡ Penalized Linear Regression.

bfun = This parameter specifies the type of the basis function. The package supports two types of basis functions:

'POLY' ≡ Polynomial Basis function

'TRIG' ≡ Trigonometric basis function.

msel = This parameter specifies the model selection criteria. The package provides for the following Model Selection criteria:

'FPE'=Final Prediction Error.
'SC'=Schwartz Criterion.
'GCV'=Generalized cross-validation.
'SMS'=Shibata's model selector.

‘VM’=Vapnik’s Measure.
‘XVAL’= Leave-one-out Cross Validation.
‘MVAL’=M-Fold Cross Validation.
‘COMP’= Comparison of different models on some pre-specified data generator.

degr = This parameter specifies the maximum degree for the basis function.

OUTPUT

model= This is a structure which contains 3 parameters for the model.

model.X= Scaled values of X in the range of [0,1]

model.Y=Scaled values of Y in the range of[-1,1]

model.coef=Array of the model parameters.

mse=(Mean Squared Error) This parameter provides the model performance on the testdata provided

EXAMPLE (LIN/PLIN REGRESSION INTERFACE)

1. Define training data set:

```
tmp =load('train.txt','-ascii');     %This is a 99x2 matrix
trn_data.X = tmp(:,1);                 %Load the X value of the structure trn_data
trn_data.Y = tmp(:,2);                %Load the Y value of the structure trn_data
```

The trn_data will look something like:

```
display(trn_data);
trn_data =
                    X: [99x1 double]
                    Y: [99x1 double]
```

2. Similarly define the Test data set:

```
tmp=load('test.txt','-ascii');
tst_data.X=tmp(:,1);
tst_data.Y = tmp(:,2);
```

3. Now perform the regression:

```
[model,mse]=regrIntf(trn_data,tst_data,'LIN','TRIG','VM',10);
```

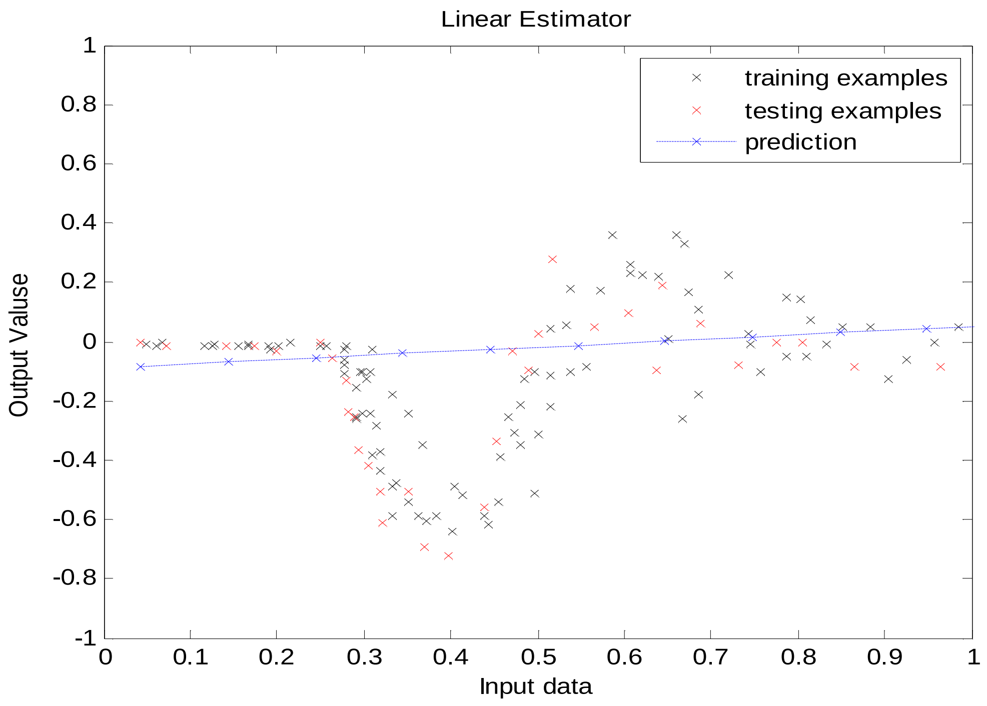
   Here we used rtype=Linear Regression
   bfun=Trigonometric Function
   msel= Vapnik's Measure
   degr=10

4. So now we have the optimum model parameters (based on VM)in the model data structure.

```
display(model);
model =
        X: [0.0435 0.1441 0.2446 0.3452 0.4458 0.5464 0.6469 0.7475 0.8481  0.9487 1.0492]
        Y:  [-0.0827  -0.0687  -0.0546  -0.0406  -0.0266  -0.0126  0.0015  0.0155  0.0295  0.0435
        0.0575]
         coef: [0.1394 -0.0887]
```

5. In order to display the model output in the scatter plot of the training and test samples use the **plotregr1D** routine.

```
plotregr1D(traindata,testdata,model);
```



So now you can see the performance of the model in a graphical form. You can also check the Mean Squared error by typing mse in the Command Window.

EXAMPLE 2: Here we use the regrIntf to Compare different Model selection criteria

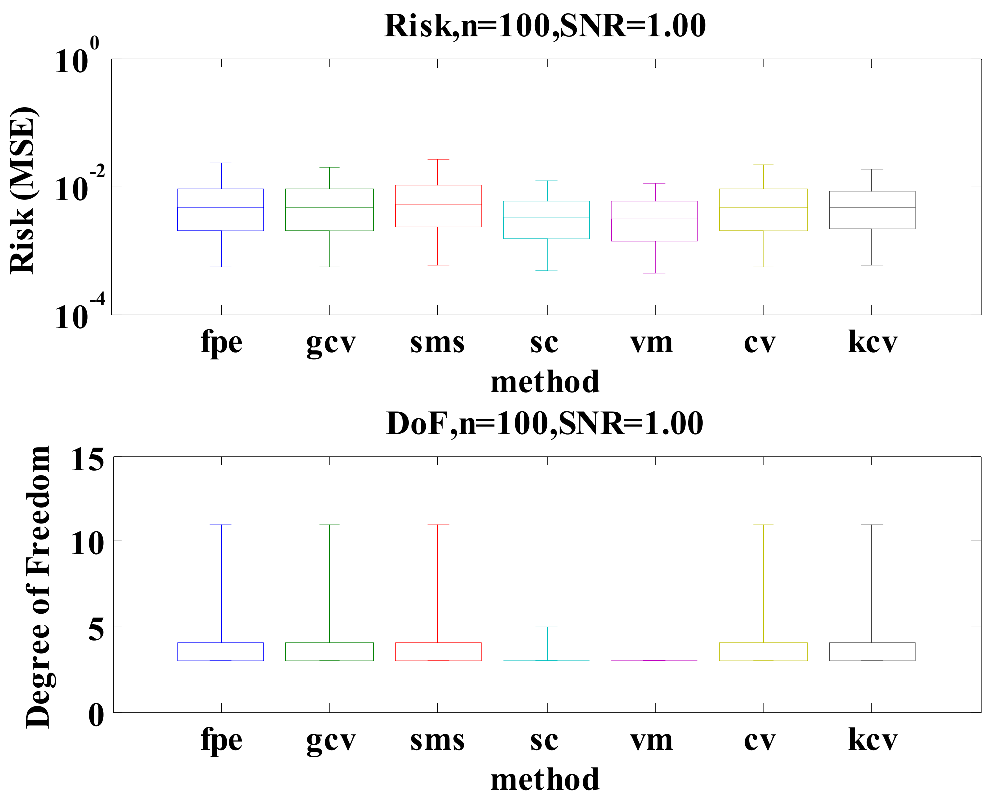
This functionality can be used to compare the performance of different models on some synthetic data generating function predefined by the software.

For this the steps that need to be performed are:

1. Provide the Train data and the Test data as specified before.
2. Just replace the msel = ‘COMP’ and use the regrIntf as shown below:

   ```
   [model,mse]=regrIntf(trn_data,tst_data,'LIN','TRIG','COMP',10);
   ```

3. In this case new training and test sets will be generated using some internal function generator for 50 times and a comparison will be provided for the different model selection criteria on the basis of the Mean squared Error and the Complexity parameter of the Regression type.



For E.g.: Using the above function we get the graphical display of the comparison for different models using Box plot diagrams.

5. Limitations:

There are a number of limitations set up on this package.

1. While using the single regression routines the outputs are always scaled and plotted in the range of [0,1] for the X values and in the range of [-1,1] for the Y values.
2. The regression functions can only be used for a maximum input dimension of 2.
3. The plotregr1D can be used to plot only one-dimensional input/output mapping.
4. While using the comparison functionality the data generator function is predefined by the software package. No option is given to the users to specify their own generator functions.

ADAPTIVE METHODS (XTAL)

XTAL is mainly designed for a basic comparison of different regression methods evaluated in the Predictive learning Framework. This package mainly aims toward providing a comprehendible interface for the naïve users.

INTRODUCTION to XTAL

XTAL contains five representative methods.

1. Multi-Layer Perceptron. (ANN1): The XTAL package uses a version of multilayer feed forward networks with a single hidden layer described in Masters (1993). This version employs conjugate gradient descent for estimating model parameters (weights) and performs a very thorough (internal) optimization via simulated annealing to escape from local minima (10 annealing cycles). The original implementation from Masters (1993) is used with minor modifications. The method's implementation in XTAL has a single user-defined parameter - the number of hidden units. This is the complexity parameter of the method.

2. Multivariate Adaptive Regression Splines. (MRS1): The original code provided by J. Friedman is used (Friedman 1991). In the XTAL implementation the user selects the maximum number of basis functions and the adaptive correction factor. The interaction degree is defaulted to allow all interactions.

3. Projection Pursuit Regression. (PPR1): The original implementation of projection pursuit, called SMART (smooth multiple additive regression technique; Friedman 1984a), was used. In addition the SMART package allows the user to control the thoroughness of optimization. In the XTAL implementation, this is set to the highest level.

4. Constrained Topological Mapping. (CTM1): The batch CTM software is used (Mulier 1994). When used with XTAL, the user supplies the model complexity penalty, an integer from 0 to 9 (maximum smoothing) and the dimensionality of the map.

5. K nearest Neighbor Regression. (KNN1): A simple non adaptive version with parameter selected by the user.For a brief description of the above methods please refer to [1]

USER INTERFACE

The main user interface of the XTAL is :

[ypred ,nrms,rms0,nmax] = xtal (trn_data, tst_data, method, params)

- **INPUT**

  trn_data: This is the training data to be provided by the user. It is a data structure having the following parameters:

  trn_data.X= a DxN matrix of the input independent variables.

  where,

  D=Dimension of the INPUT variable.

  N= Total Number of Samples.

  trn_data.y= an Nx1 matrix of the output target values.

  tst_data: The parameterization is the same as trn_data except that this data will be used for testing the model.

- **PARAMETER TUNING**

  methods: This reflects the methods that we intend to test. The xtal supports 5 types of methods:

  1. Multi-Layer Perceptron. (‘ANN1’).
  2. Multivariate Adaptive Regression Splines. (‘MRS1’).
  3. Projection Pursuit Regression. (‘PPR1’)
  4. Constrained Topological Mapping. (‘CTM1’)
  5. K nearest Neighbor Regression. (‘KNN1’)

  …….and soon enough ‘RBF1’

So when we enter the values for this parameter we use either of:

‘ANN1’,’MRS1’,’PPR1’,’CTM1’,’KNN1’ in the same format as specified.(The package is case sensitive and as such may throw an error if lower case are used).

params: These are the parameters that need to be specified for the different methods used. A brief description of the parameters for the different methods is provided below.

ANN1

This method has one parameter. This parameter sets the number of hidden neurons to be used in a three layer neural network. Greater the number of neurons the greater the training time. Values are typically between 2 to 40 but can be as high as 1000.

MRS1

This method has two principal tuning parameters.

Parameter1: is the maximum number of basis functions to use. This parameter controls the maximum amount of complexity of the model. In most cases, the results are not very sensitive to this parameter. The range of this parameter is from **1 to 100**. For most problems, this parameter can be set to a high value (50-100) which results in a good fit.

Parameter2: The second parameter is the amount of penalty given to complex models. This is an integer from **0-9** where 0 means a small penalty is given to complex models and 9 means a large penalty is given. This parameter is adjusted based on the estimated amount of noise in the data, and has the largest effect on the results. For problems with little noise a value less than 5 are suggested. For noisy problems, values larger than 5 can be used. A good starting value for this parameter is 5.

PPR1

This method requires one parameter. This parameter controls the complexity (number of terms) of the model. For most problems, a value between 1 and 10 works well. This method does not use any internal parameter selection, so this parameter directly affects the complexity of the model and must be chosen with care.

CTM1

This method requires two parameters.

Parameter1: is the number of dimensions used in the map. This parameter is chosen based on whether constraints exit between the independent variables. It should reflect the estimated intrinsic dimensionality of the data. For most problems, this is chosen to be 1, 2, or 3. This should always be less than or equal to the dimension of the input space.

Parameter2: The second parameter is a smoothing parameter. This is an integer from 0 to 9, where 9 indicate the smoothest model. If this parameter is set to 0 then the algorithm tries to minimize the RMS error on the training set. If this parameter is set to

9, then the algorithm tries to minimize the cross-validation error. For any value between 0 and 9, a mixture of the two error measures is minimized. This parameter becomes critical for problems with high noise and/or small number of samples in the training set. For problems with low noise and large number of samples, the parameter has little effect. For high noise problems, a larger number is suggested.

KNN1
This method requires one parameter called "k" which specifies the numbers of nearest neighbors that are averaged to form an estimate. The value for k must be greater than 0 but less than the number of samples in training file. The package supports a **maximum value of K=100.**

(**Note**: The maximum values and the ranges of the parameters should not be exceeded in any case. Though the package does not block setting of such out of range values, but it causes erroneous values in such cases).

- **OUTPUT**
  ypred: This is the predicted output of the tst_data provided. This is an array of dimension [Nx1]
  where, N= Total number of Samples.

  nrms: This is the Normalized RMS error obtained on the Test Set. This is obtained by dividing the RMS error on the test set data by the estimated Standard deviation of the test set.

  rms0: The Standard deviation of the y values of the test set data.

  nmax: This is obtained by dividing the absolute maximum error on the test data by the range of y values of the test data.

Here we will show two examples for the XTAL usage. For further reference please see[1] and [2].

EXAMPLE (XTAL)

Example 1: (1D Regression) In this example we shall use the motorcycle data http://www.quantlet.de/mdstat/scripts/anr/html/anrhtmlframe139.html. For the sake of simplicity we construct the validation set by chaffing out every 5th Sample from the data set and using the rest of the data for training. We shall consider the following methods 'ANN1','MRS1','PPR1',CTM1 and 'KNN1'.The range of values that we shall select for the different methods are:
'ANN1'= [5; 10; 25; 50; 100]
'MRS1'=[[20,0]; [20,5]; [20,9]; [50,0]; [50,5]; [50,9]; [100,0];[100,5];[100,9]];
'PPR1'= [1; 5; 10; 50];
'CTM1'= [[1, 0]; [1, 2]; [1, 5]; [1, 9]];
'KNN1'= [2; 5; 8; 11];
The main intent of this example is to highlight the usage of XTAL.

We shall approach the problem stepwise.

STEP 1: Load the data and create the train/test data structures.

```
% load training/testing data
train=csvread('motorcycle_train.csv');
test=csvread('motorcycle_test.csv');

trn_data.X=train(:,1)';
trn_data.y=train(:,2);

tst_data.X=test(:,1)';
tst_data.y=test(:,2);
```

STEP 2: Define the methods and their parameters over which the test has to be done.

```
exp(1).method = 'ANN1';
exp(1).params = [5; 10; 25; 50; 100]

exp(2).method = 'MRS1';
exp(2).params = [[20,0]; [20,5]; [20,9]; [50,0]; [50,5]; [50,9]; [100,0];[100,5];[100,9]];

exp(3).method = 'PPR1';
exp(3).params = [1; 5; 10; 50];

exp(4).method = 'CTM1';
exp(4).params = [[1,0];[1,2];[1,5];[1,9]];
exp(5).method = 'KNN1';
exp(5).params = [2;5;8;11];
```

STEP 3: Iteratively call the different methods and select their best model based on the performance on the test data.

```
for i=1:length(exp)

   fprintf('\nExperiment: %s\n', exp(i).method);
   fprintf('--------------------------------------\n');

   [ypred,exp(i).nrms,exp(i).rms0,exp(i).nmax]                                    =
xtal(trn_data,tst_data,exp(i).method,exp(i).params);

   [exp(i).min_nrms,exp(i).best_param_idx] = min(exp(i).nrms);
   fprintf('Minimal NRMS=%f achieved with parameter ( ', exp(i).min_nrms);
   fprintf('%d ', exp(i).params(exp(i).best_param_idx,:));
   fprintf(')\n');
   h0=figure;
   set(h0,'Name',exp(i).method);
   h1=plot(trn_data.X,trn_data.y,'kx');
   hold on;
   h2=plot(tst_data.X,tst_data.y,'rx');
   h3=plot(tst_data.X,ypred(exp(i).best_param_idx,:),'bx--');
   legend([h1 h2 h3],'traing examples','testing examples','prediction');
end
```

STEP 4: Now select the best method based on their best model’s performance on the Test data.

```
[min_nrms,best_method] = min([exp.min_nrms]);
fprintf('\n** Summary **\n\n');
fprintf('Method        NRMS        nmax  parameter\n');
fprintf('-------------------------------------------------\n');
for i=1:length(exp)
   fprintf('%s   %12.6f %12.6f  ( ',...
         exp(i).method,exp(i).min_nrms,exp(i).nmax(exp(i).best_param_idx));
   fprintf('%d ', exp(i).params(exp(i).best_param_idx,:));
   fprintf(')');
   if i == best_method
      fprintf('        best result');
   end
   fprintf('\n');
end
```

OUTPUT(Graphical)

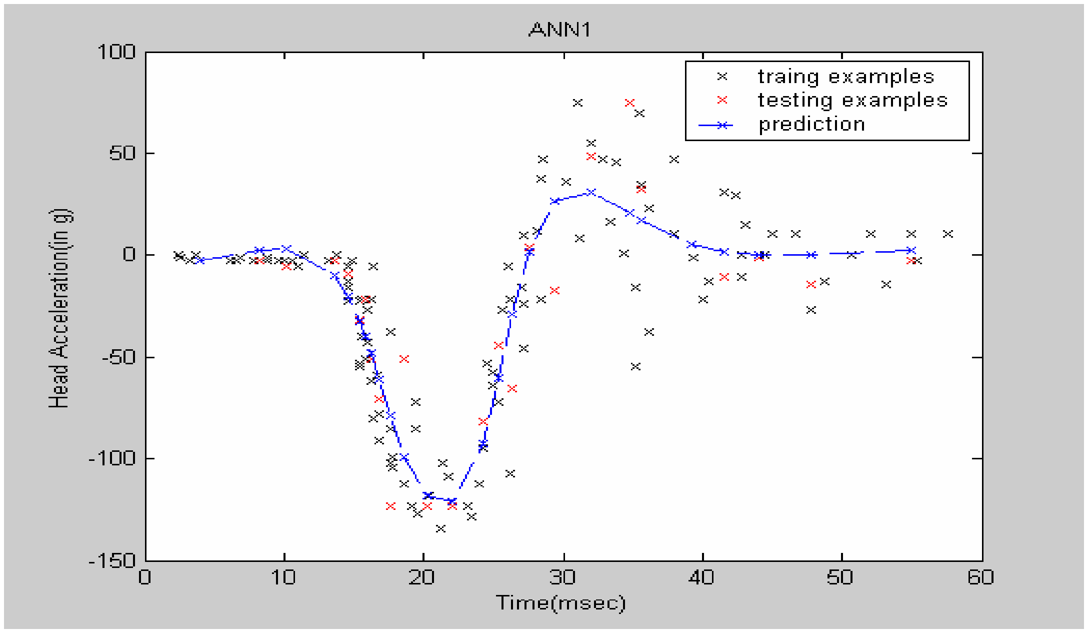


Fig1: Best Model for ANN with Hidden number of units=5

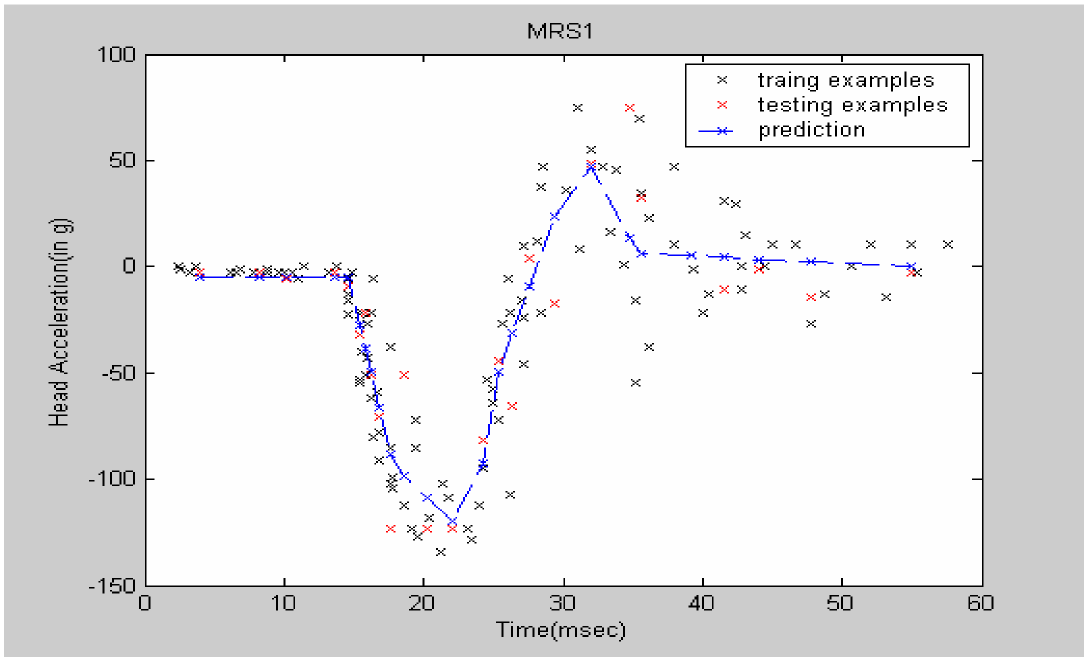


Fig2: Best model for MARS with parameter (20,1)

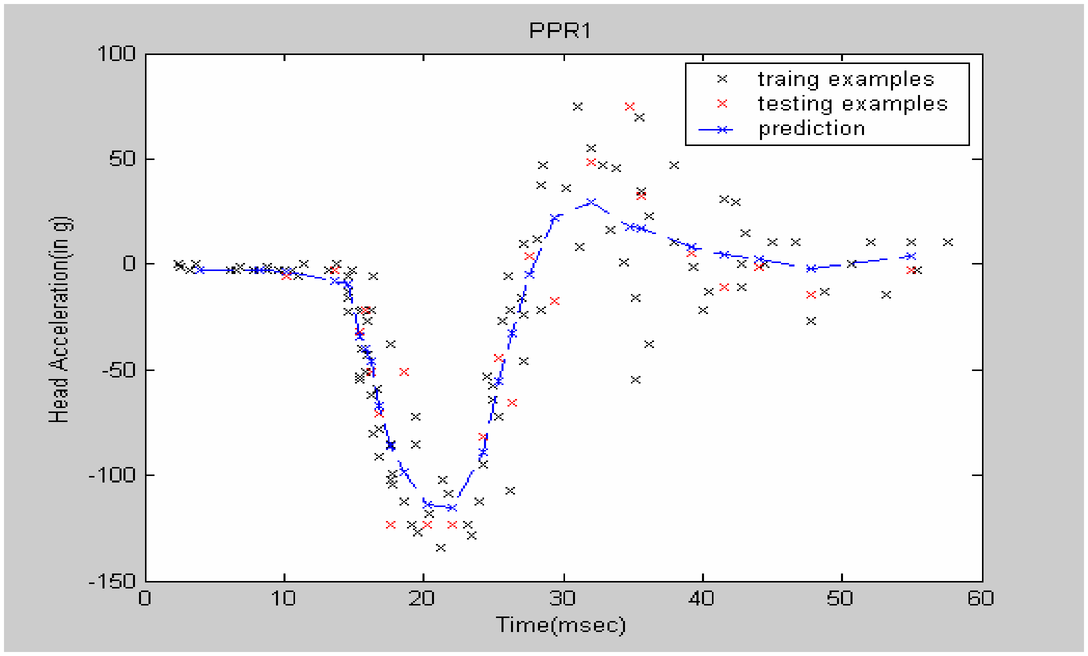


Fig 3: Best Model for PPR with number of units=5. (This is also the best method)

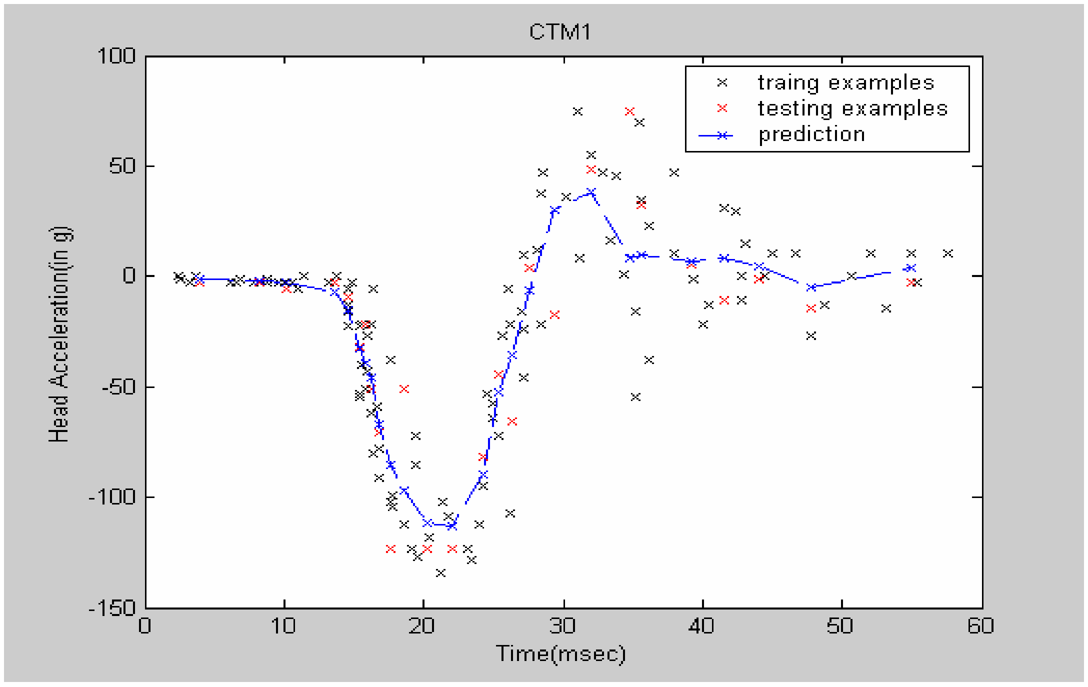


Fig4: Best model for CTM with parameters (1, 9) (makes sense)

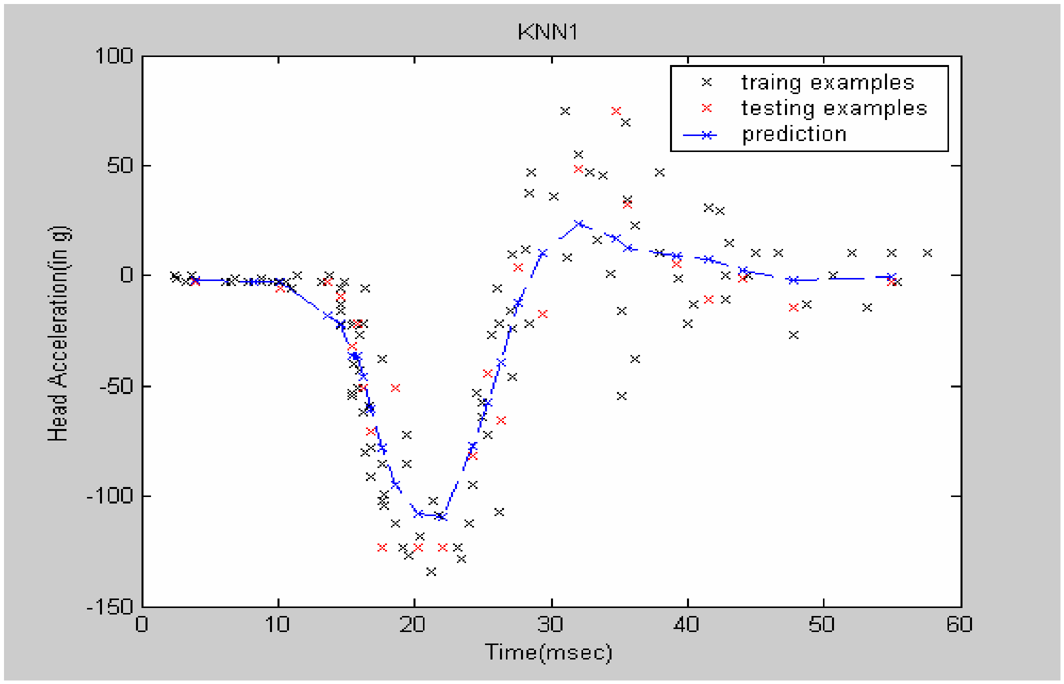


Fig5: Best model for KNN with value of K=11

Moreover XTAL also provides the summary of performance of the best models of the different methods.

**** Summary ****

Method	NRMS	nmax	parameter	
ANN1	0.459831	0.223371	(5)	
MRS1	0.455323	0.206250	(20 0)	
PPR1	0.440387	0.199387	(5)	best result
CTM1	0.470975	0.238446	(1 9)	
KNN1	0.450871	0.229223	(11)	

EXAMPLE 2: (High Dimensional Data) In this example we shall see further usage of XTAL for the Computer Hardware data. This is a publicly available at: http://archive.ics.uci.edu/ml/datasets/Computer+Hardware

The data set has 209 samples and 6 attributes. For this example we perform a 5 Fold Cross Validation to select the best model. The methods and parameters used in this case are given below:

'ANN1' = [5; 10; 25; 50; 100]

'MRS1' = [[20,0]; [20,5]; [20,9]; [50,0]; [50,5]; [50,9]; [100,0];[100,5];[100,9]];

'PPR1'= [1; 5; 10; 50];

'CTM1'= [[1,0];[1,2];[1,5];[1,9];[2,0];[2,2];[2,5];[2,9];[3,0];[3,2];[3,5];[3,9]];

'KNN1' with K=1, 2, 3, 4… 50

Here we show the usage of XTAL in a stepwise manner.

STEP 1: LOAD the Data

We do this in the Cross Validation part.

STEP 2: DEFINE the Methods and Parameters

```
exp(1).method = 'ANN1';
exp(1).params = [5; 10; 25; 50; 100]

exp(2).method = 'MRS1';
exp(2).params = [[20,0]; [20,5]; [20,9]; [50,0]; [50,5]; [50,9]; [100,0];[100,5];[100,9]];

exp(3).method = 'PPR1';
exp(3).params = [1; 5; 10; 50];

exp(4).method = 'CTM1';
exp(4).params = [[1,0];[1,2];[1,5];[1,9];[2,0];[2,2];[2,5];[2,9];[3,0];[3,2];[3,5];[3,9]];

K=1:50;
exp(5).method = 'KNN1';
exp(5).params = K';
```

STEP 3: Find the Best Model for the different methods based on the performance on the Cross Validation error.

```
for i=1:length(exp)

   fprintf('\nExperiment: %s\n', exp(i).method);
   fprintf('----------------------------------------\n');
   nrms_av=zeros(size(exp(i).params,1),5);
   for j=1:5
```

```
    [trn_data,tst_data]=crossValFold(j);                    %Check the CV is done here
    [ypred,nrms,rms0] = xtal(trn_data,tst_data,exp(i).method,exp(i).params);
    nrms_av(:,j)=nrms_av(:,j)+nrms;
    end
    exp(i).nrms=mean(nrms,2);
    [exp(i).min_nrms,exp(i).best_param_idx] = min(exp(i).nrms);  %Select the best model

    fprintf('Minimal nrms=%f achieved with parameter ( ', exp(i).min_nrms);
    fprintf('%d ', exp(i).params(exp(i).best_param_idx,:));
    fprintf(')\n');
end
```

STEP 4:Select the best Method

```
[min_nrms,best_method] = min([exp.min_nrms]);
fprintf('\n** Summary **\n\n');
fprintf('Method       nrms       nmax  parameter\n');
fprintf('-------------------------------------------------\n');
for i=1:length(exp)
    fprintf('%s   %12.6f ( ',...
            exp(i).method,exp(i).min_nrms);
    fprintf('%d ', exp(i).params(exp(i).best_param_idx,:));
    fprintf(')');
    if i == best_method
        fprintf('       best result');
    end
    fprintf('\n');
end
```

OUTPUT: No Graphical representation can be given in this case. However the performance summary can be obtained as:-

```
** Summary **
Method       nrms       nmax  parameter
-------------------------------------------------
ANN1      0.538486 ( 5 )
MRS1      0.550195 ( 20 9 )
PPR1     0.438475 ( 5 )       best result
CTM1      0.846139 ( 3 2 )
KNN1      0.467276 ( 1 )
```

SECTION 2: OTHER SOFTWARES (BRIEF OVERVIEW)

In this section we shall simply provide a brief overview of the different softwares available. This report will simply contain the general Framework of the packages and the strength and the weakness of such a Framework. The main reason behind such an approach is that most of these softwares have their own help documents and as such the author sees no credit in replicating most of those materials. However the reader is welcome to contact the author of the document for any help regarding the softwares that will be covered in this section.

PRTools:(GENERAL FRAMEWORK)

Publicly available at: http://www.prtools.org/

In such a framework we mainly identify two types of interface.

1. **Interface1:** Interface to determine the model.
2. **Interface2:** Interface to predict the output on some test data.

1. **Interface1:** This is provided in the form
   [model, Error] = method (trainset, parameters)

- **INPUT:**
  trainset: This is an object of type data. The package provides another interface
  trainset =data(X, Y) to convert the input data to the following object.

- **TUNABLE PARAMETERS**
  parameters: These are generally some vector of parameters. This mainly depends upon the type of the method under consideration.

- **OUTPUT**
  model: This is the model obtained by the method on the trainset. This is an object of class mapping.
  Error: This is the Error(or model performance) on the trainset.

2. **Interface2:**These kinds of interfaces are provided in the form:
   Ypred=testset*model

As we can see the main intent of this kind of object oriented programming is to overload the operators. Here we see a good example for this. However we see a number of issues in such kind of framework.

GOOD

- The Object Oriented Programming concept may seem very helpful to the advanced users where the level of abstraction is quite high.
- Unlike STPRTool(especially the Quadratic Decision Boundary) the usage of interfaces are really easy. The user's can just plug in the Training data and expect the equivalent decision boundary without any form of mapping from input space to Output space etc.

BAD

- No 'get_param' method has been provided. In such a case the user will not have the view ability of the model parameters. Infact a basic point for every object oriented programming is that it should have a 'get' and a 'set' method.
- User need to convert the object to a structure to get the parameters. This is seen as an added overburden upon the user.

Moreover upon private communication with the designers it has been confirmed that the Toolbox is mainly intended for Classification problems. The toolbox does have some interfaces for the regression type of problems, but they are not fully functional.

WEKA (GENERAL FRAMEWORK)

Here we shall be referring to the Weka 3.5.8.This is publicly available at:
http://www.cs.waikato.ac.nz/ml/weka/
Weka is a collection of machine learning algorithms for data mining tasks. In the Version 3.5.8 the organization of WEKA class hierarchy has changed from the previous versions. The Main class is no more the GUI Chooser. Of course in WEKA there can be a number of different types of Windows; however the main frame work is the same. In this report we shall simply provide the basic guideline about its usage.(The Examples will be shown mainly for the Explorer Window)

We mainly identify WEKA's framework as

- **INPUT**
  In this case the data is presented as a .csv or .arff file. (There are a number of other file formats). These data can be loaded in the interface and can be preprocessed by a number of filters provided.

- **METHOD/PARAMETER SELECTION**
  The WEKA mainly identifies all kinds of problems either as
  - Classify: It seems rather confusing to naïve users who may be unaware that WEKA provides the different regression methods in the Classify tab!
  - Clustering
  - Association

- **OUTPUT**
  An Output window provides the user with the Model parameters and the Model performance. One more issue with the WEKA is that it does not provide any graphical display of the Decision boundary. We can simply presents a form of representation of the decision boundary and leaves the rest of imagination to be done by the user!

EXAMPLE(WEKA): For the sake of simplicity we use the Example 2 in pg no .Unlike XTAL ,WEKA does not provide some of the methods like PPR, MARS, CTM (even KNN regression),however it does provide the MLP function.

We test the CPU Hardware data for the set of Parameter setting

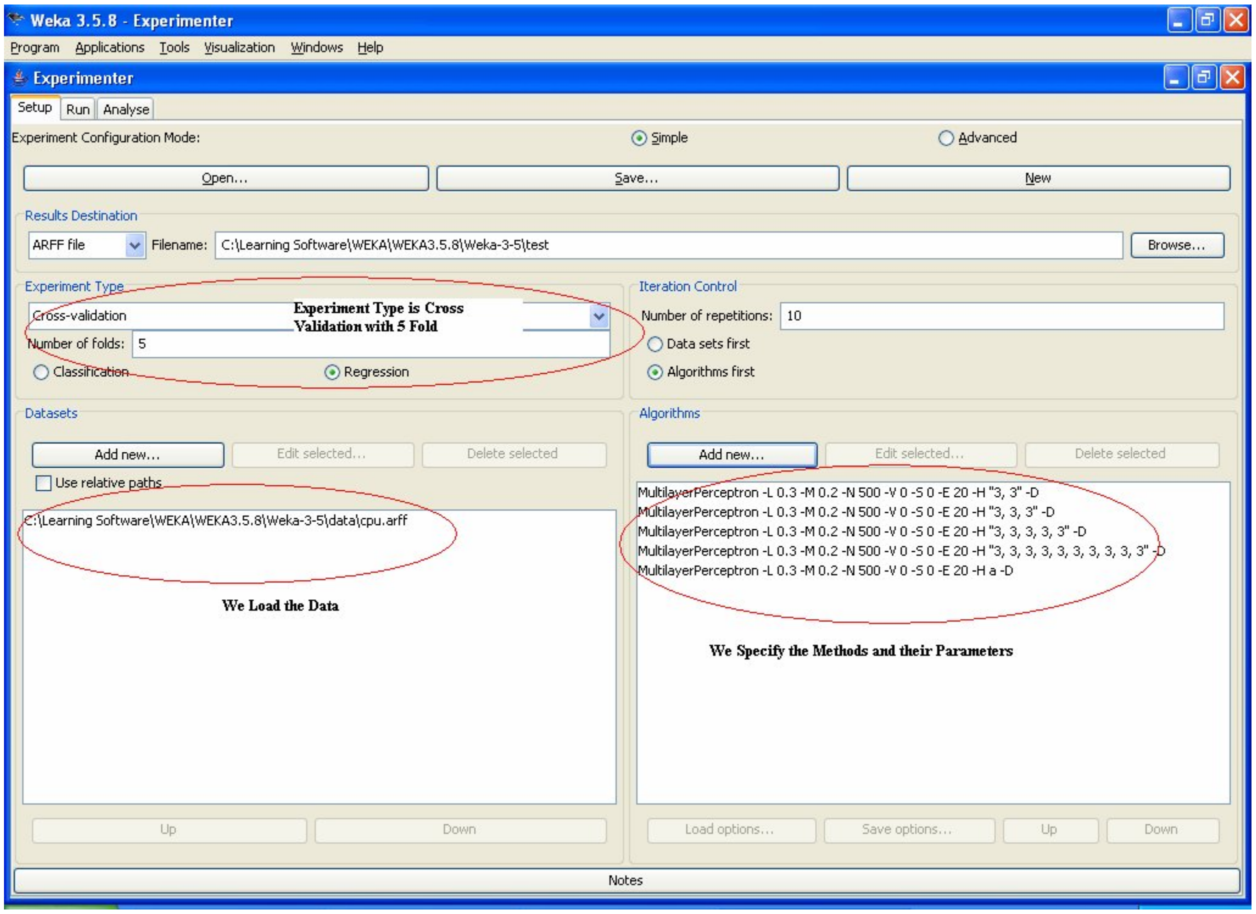
Variable Learning Rate=True;
%In this case the Learning rate is updated by dividing by the Epoch number.

Number of Hidden Layers=
Case 1: 2 Hidden layers with 5 Neurons per layer
Case 2: 3 Hidden Layers with 5 Neurons per layer.
Case 3: 5 Hidden layers with 5 Neurons per layer.
Case 4:10 Hidden layers with 5 Neurons per layer.
Case 5: We keep it as 'a' =(attribs + classes) / 2.This is a wild card entry. A number of other wild card entries are supported.
In this experiment the Initial Learning rate is selected as 0.3 and the Momentum term is kept 0.2.Moreover the training time is kept as 500 epochs.

We shall search for the best model based on the RMS error on the 5 Fold Cross Validation

STEP 1:

Load the data in the Experimenter tool.


Weka 3.5.8 - Experimenter

Experimenter

Experiment Configuration Mode: [✓ Simple] Advanced

Results Destination: ARFF file   Filename: C:\Learning Software\WEKA\WEKA3.5.8\Weka-3-5\test

Experiment Type: Cross-validation

Number of folds: 5

Classification [✓ Regression]

Experiment Type is Cross Validation with 5 Fold

Iteration Control: Number of repetitions: 10

Data sets first [✓ Algorithms first]

Datasets:

```
C:\Learning Software\WEKA\WEKA3.5.8\Weka-3-5\data\cpu.arff
```

We Load the Data

Algorithms:

```
MultilayerPerceptron -L 0.3 -M 0.2 -N 500 -V 0 -S 0 -E 20 -H "3, 3" -D
MultilayerPerceptron -L 0.3 -M 0.2 -N 500 -V 0 -S 0 -E 20 -H "3, 3, 3" -D
MultilayerPerceptron -L 0.3 -M 0.2 -N 500 -V 0 -S 0 -E 20 -H "3, 3, 3, 3, 3" -D
MultilayerPerceptron -L 0.3 -M 0.2 -N 500 -V 0 -S 0 -E 20 -H "3, 3, 3, 3, 3, 3, 3, 3, 3, 3" -D
MultilayerPerceptron -L 0.3 -M 0.2 -N 500 -V 0 -S 0 -E 20 -H a -D
```

We Specify the Methods and their Parameters

Program Applications Tools Visualization Windows Help
Setup Run Analyse
Open... Save... New
Browse...
Add new... Edit selected... Delete selected
Use relative paths
Up Down
Load options... Save options...
Notes

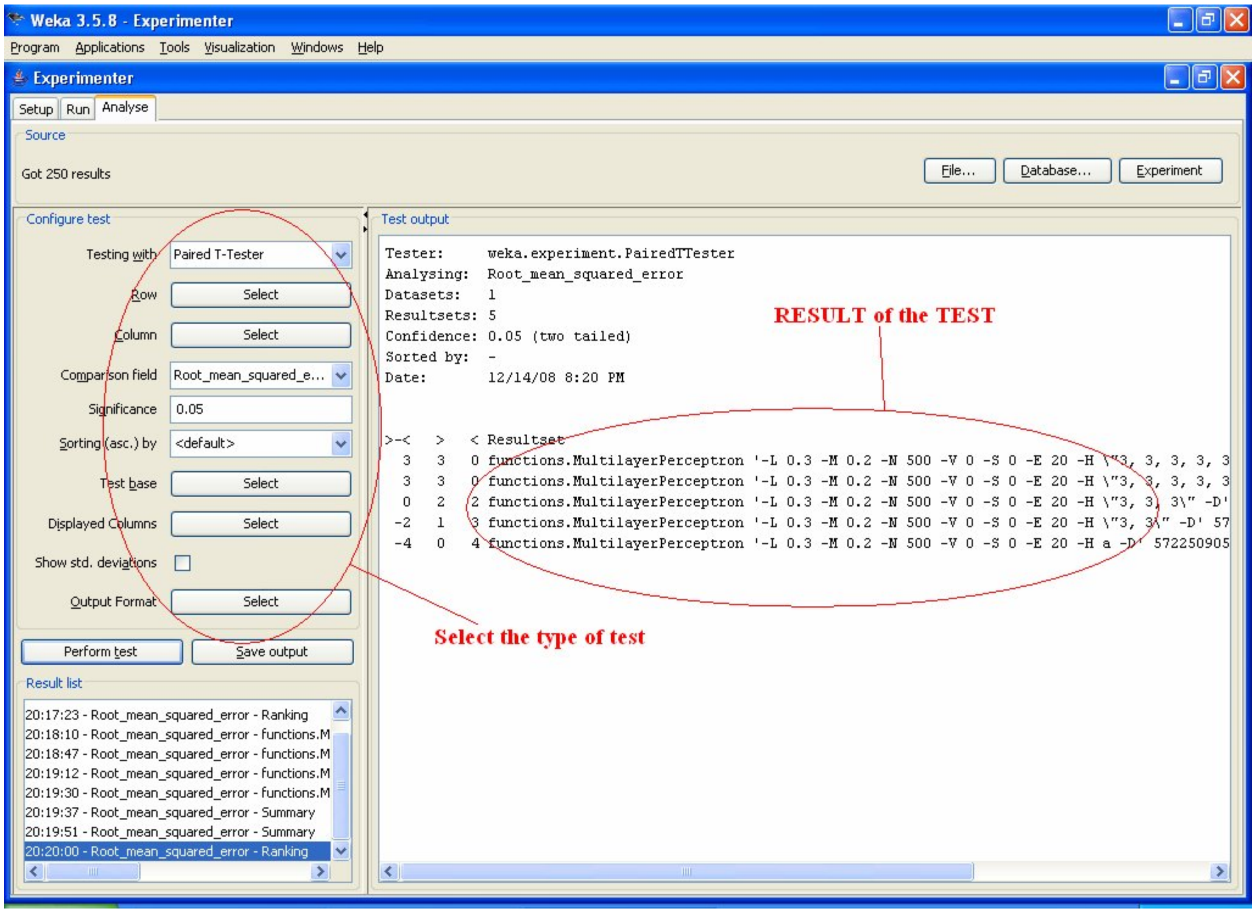
As we can see the left Lower Box we load the data set. In the Left upper box we specify the type of test to be performed. In the right lower box we specify the methods and their parameters.

STEP 2

RUN all the methods. We do not show it here. It is simply clicking the RUN Button in the next Tab. The Run Window moreover displays the status of the execution of the experiment.

STEP 3

We can analyze the methods on a number of different statistics. This can be done in the analyze window.



Got 250 results

Testing with: Paired T-Tester
Row: Select
Column: Select
Comparison field: Root_mean_squared_e...
Significance: 0.05
Sorting (asc.) by: <default>
Test base: Select
Displayed Columns: Select
Show std. deviations: ☐
Output Format: Select

```
20:17:23 - Root_mean_squared_error - Ranking
20:18:10 - Root_mean_squared_error - functions.M
20:18:47 - Root_mean_squared_error - functions.M
20:19:12 - Root_mean_squared_error - functions.M
20:19:30 - Root_mean_squared_error - functions.M
20:19:37 - Root_mean_squared_error - Summary
20:19:51 - Root_mean_squared_error - Summary
20:20:00 - Root_mean_squared_error - Ranking
```

```
Tester:     weka.experiment.PairedTTester
Analysing:  Root_mean_squared_error
Datasets:   1
Resultsets: 5
Confidence: 0.05 (two tailed)
Sorted by:  -
Date:       12/14/08 8:20 PM


>-<   >   < Resultset
  3   3   0 functions.MultilayerPerceptron '-L 0.3 -M 0.2 -N 500 -V 0 -S 0 -E 20 -H \"3, 3, 3, 3, 3
  3   3   0 functions.MultilayerPerceptron '-L 0.3 -M 0.2 -N 500 -V 0 -S 0 -E 20 -H \"3, 3, 3, 3, 3
  0   2   2 functions.MultilayerPerceptron '-L 0.3 -M 0.2 -N 500 -V 0 -S 0 -E 20 -H \"3, 3, 3\" -D'
 -2   1   3 functions.MultilayerPerceptron '-L 0.3 -M 0.2 -N 500 -V 0 -S 0 -E 20 -H \"3, 3\" -D' 57
 -4   0   4 functions.MultilayerPerceptron '-L 0.3 -M 0.2 -N 500 -V 0 -S 0 -E 20 -H a -D' 572250905
```

As we can see the user has a number of options for selection of the test type.

GOOD

- WEKA provides a repository of Methods that no other software can match.
- It is totally graphical and easy to use.

BAD

- WEKA provides the Regression routines in the Classify tab. Any naïve(as well as experienced user) is likely to get confused!
- The visualize window for WEKA does not provide any decision boundary and as such the users are left to the qualms of their imagination capability.
- Although WEKA does provide a lot of flexibility in parameter selection. It does not provide any guidelines for parameter selection. It seems really hard for a user to decide upon the parameter values that they might need to select for any particular method. In short, the parameter tuning in WEKA though flexible lacks appropriate guidance.

SPIDER

The spider an object orientated environment for machine learning in Matlab. Aside from easy use of base learning algorithms, algorithms can be plugged together and can be compared with, e.g model selection, statistical tests and visual plots. This gives all the power of objects (reusability, plug together, share code) but also all the power of Matlab for machine learning research.

In a sense SPIDER is quite similar to PRTool with the exception that SPIDER provides the get and the set methods. The basic

GENERAL FRAMEWORK

This software has by far the most complex framework. Here everything is an object of different classes. So any new user is likely to get confused with the different jargons that it uses. Moreover the help documents are not at par with the level of coding in this software. This software has the best level of abstraction but this comes with the worst viewablity.

- **INPUT:**
  d=data(X,Y)
  For providing the input we need to construct an object of class data. The elements of the object are the X and Y values of the trainset.

- **PARAMETER TUNING**
  This is also totally dependent upon the type of method used. A general form of such parameter tuning is:
  Select the method
  ```
  alg=algorithm;                     %This is the type of Algorithm used.
  ```

  TUNE the parameters for the method
  ```
  alg.parameters=values;             %Update the present algorithm object
  or
  alg2=param(alg,hyperparameters);   %Create a new object
  ```
- **OUTPUT**
  A number of output methods are defined,viz,
  test, loss, plot etc

EXAMPLE: In this case we repeat the EXAMPLE provided in pg .We present that using SPIDER is almost as powerful as the evalsvm interface provided by the STPRTool. We provide a step wise illustration.

STEP 1 : Load the data.

```
d=data(X,Y);                              %Here d is an object of class data
```

STEP 2: Define the method and perform parameter tuning

```
s=svm;                                    %Define an object of class svm

s.child=kernel('rbf',1);                  %Specify the kernel type
 s_par=param(s,'rbf',[0.1,0.5,1,5]);      %Specify the range of parameters
s_CV=cv(s_par);
s_CV.folds=15;                            %Specify the Cross Validation Object
[err model]=train(s_CV,d);                %Train the Model
loss(err);                                %Find the Loss
```

GOOD

- As we can see in this case we can easily test a model for a number of C values and for a range of algorithm hyperparameters. So such a framework is very powerful.

BAD

- As viewable in the previous example.The user has no viewability of the model.
- No graphical option has been provided for the model object of class CV. The user has to search for the best model and then plot the model.
- It has a number of Naming conflict with the basic MATLAB Toolbox. The perl script for correcting this conflict does not work!

SECTION 3: SOME RECOMMENDATIONS

In this section we provide some recommendation for the different types of problems. This is provided in the table below:-

ID	Problem Type	Recommended Software Package
1	Clustering/Dimensionality Reduction	PRTools/SOM(EE 8591)
2	Classification	STPRTool
3	Regression	XTAL
4	SVM(Classifier)	STPRTool
5	SVM(Regression)	LIBSVM

This Recommendation is aimed towards presenting the simplest interface with genuine inherent algorithms. The user may feel free to deviate from this recommendation.

CONCLUSION

In this Project we provide a somewhat detailed overview of the Softwares present in the Course website and a brief overview of some of the most prevalent Machine learning softwares available. Our focus has mainly been towards presenting the best(simple) softwares . The ulterior goal of the project has been to quarry the different interfaces available in the different software packages and finally to present with the best available interface for implementing any method.

FUTURE SCOPE

This project can be further improved in many directions. We simply provide some of them that we feel should enhance the essence of the work.

1. This document only focuses on the basic usage of the different software available in the course website. A detailed usage considering a wide range of problems starting from different types of synthetic data to real data may be considered. In that way the pros and cons of the softwares can be explored to even greater details.

2. More softwares need to be considered. Moreover the author understands that the general framework provided for the other softwares may seem confusing to new users. But it is

also to be noted that a balance has to be maintained between the amount of material covered and the size of the document.

Reference:

[1] Vladimir Cherkassky and Filip Mulier, 'LEARNING FROM DATA',2nd Edition, John Wiley and Sons. Inc 2007

[2] Vladimir Cherkassky,Don Gehring and Filip Mulier , '***Comparison of Adaptive Methods for Function Estimation from Samples'***, IEEE Transactions on Neural Networks,Vol.7,No 4,JULY 1996.