COMP3702/7702 Assignment 2, 2010

Handwritten letter recognition

SU Sheng Loong (42397997) and TEE Lip Jian (42430942)

Parts completed:

1A. Single-layer Neural Network (2)

1B. ID3 Decision Tree (2)

2A. Multi-layer Neural Network (5)

2B. ID3 Decision Tree with Pruning (5)

2C. Different Machine Learning Technique (7)

3A. Create an Ensemble of Classifiers (4)

3B. Implement a pre-processing technique (4)

3C. Overtraining prevention (4)

4. Competition (1+)

Reference (1)

This work qualifies for 35 marks.

1A. Single-layer Neural Network

Aims of the Study

To optimise and assess the performance of the already implemented single-layer neural network upon the data set by manipulating parameters of the machine learning technique.

Description on the Optimisation Investigated

The implemented single-layer neural network was optimised and evaluated for the performance of classifying handwritten characters based on the data set available. There were a couple of parameters to manipulate for optimising this machine learning technique, including learning rate, the number of iterations of learning (number of presentations) and the size of the test and training sets.

Performance on Training and Test Data

The following table shows the performance of this machine learning technique on classifying the test sets given. As shown in the table, the parameters were specified for training the classifier and were manipulated throughout the tests. Note that the number of training sets is the denominator of the training error fraction. Likewise, the number of test sets is the denominator of the test error fraction. All tests were run 10 times to obtain the average test results. In terms of implementation, some code was added to retrieve the training and test error counts.

Table 1: Basic Study: Single-layer Neural Network Optimisation

Algorithm	Description	Learning rate	Number of presentations	Training error	Test error
NN/BP	Single layer	0.1	50000	49/1000	290/1000
NN/BP	Single layer	0.2	50000	61/1000	292/1000
NN/BP	Single layer	0.3	50000	59/1000	281/1000
NN/BP	Single layer	0.1	100000	46/1000	275/1000
NN/BP	Single layer	0.1	500000	21/1000	264/1000
NN/BP	Single layer	0.1	50000	163/2000	660/2000
NN/BP	Single layer	0.1	50000	54/1000	1522/4000
NN/BP	Single layer	0.1	50000	616/4000	166/1000

It can be seen that as learning rate increases, the performance initially worsens but begins to improve later when the learning rate is set to 0.3. While large number of presentations exhibit improvement over the training and testing performance, the training process consume relatively considerate amount of time. As the number of data sets increases, the performance seems to worsen. Having smaller training sets relatively to the test sets results in much worse performance, as shown by the 1522 errors out of the 4000 test sets. In contrary, Having more training sets compared to testing sets produces better testing performance while getting poor training performance.

Conclusion

Using low learning rate, high number of iterations and same number of training and testing data sets can yield ideal performance for both training and testing. In addition, the training would be more effective with more time given for the process.

1B. ID3 Decision Tree

Aims of the Study

To optimise and assess the performance of the already implemented ID3 decision tree upon the data set by manipulating parameters for this machine learning method.

Description on the Optimisation Investigated

The already implemented ID3 decision tree has less parameter to manipulate as compared to the single-layer neural network machine learning technique. In this section, the focus point is the manipulation of the size of the training/test sets for achieving good performance of such implementation. In other words, the assessment focuses on how the performance can be affected by the difference between the size of the training sets and the size of the test sets.

Performance on Training and Test Data

The following table depicts the tests run to assess the performance of the implemented ID3 decision tree on classifying the test sets. As mentioned previously, the number of training sets is the denominator of the training error fraction and the number of test sets is the denominator of the test error fraction. Once again, all tests were run 10 times to obtain the average test results. In terms of implementation, some code was added to retrieve the training and test error counts. In addition, the original source files were modified slightly to use the ID3 decision tree for training and testing instead of the single-layer neural network.

Table 2: Basic Study: ID3 Decision Tree Optimisation

Algorithm	Description	Training error	Test error
ID3 DT	Binary tree without pruning	313/2803	1653/2803
ID3 DT	Binary tree without pruning	124/1000	2994/4606
ID3 DT	Binary tree without pruning	634/4606	358/1000
ID3 DT	Binary tree without pruning	246/2000	2319/3606
ID3 DT	Binary tree without pruning	525/3606	868/2000

Based on the results presented in the table above, it seems that same size of training and test sets do not produce satisfactory results – the first test case has 1653 test errors. This could be due to the reason that the total data sets available are not sufficient. If data sets are more than 5606, there would be more both the training and test sets and this would result in more favourable test performance.

The test cases 2 and 4 clearly show that if training sets are less than test sets, the test performance would be terrible. The reason is obvious – the classifier was undertrained, meaning the classifier did not have sufficient data sets for training.

The test cases 3 and 5 have the most decent test performance for this technique. Both test cases have more training sets than the test sets and most likely the classifiers had enough datasets for training and thus have fewer errors. That said, these two test cases have the highest number of training errors for this experiment.

Conclusion

Having more training sets seems to have higher performance on classifying the character. Unfortunately due to the limitation of the number of total datasets given, the experiment cannot show that by having more datasets, equal number of training/testing sets would yield better performance.

2A. Multi-layer Neural Network

Aims of the Study

To extend the existing neural network to multi-layer neural network to produce more advanced classifier.

To investigate the optimisation and performance of the advanced classifier obtained from such machine learning technique.

Description on the Extension of the Basic Implementation

The existing neural network implementation was extended by adding a layer of hidden units. The implementation had one hidden layer positioned in between the input units and the output units so that the input value would be feed-forwarded to the hidden units. The input value would be altered based on the weight and bias between the input layer and hidden layer before the value is feed-forwarded to the output layer. The calculations based on root-mean-squared error when comparing output values and the desired values were then propagated backwards to throughout the network including the hidden layers. The weights and biases in the network would then be changed accordingly to improve the classifier.

The implementation of the extension was done with reference to Russel and Stuart's *Artificial Intelligence, 3rd edition* (2009).

Pseudo Code of the Implementation

```
function train(inputs, desired, eta)
        returns root-mean-squared-error
        inputs: inputs, a set of input values
                desired, a set desired output values
                eta, the learning rate which is always between 0 and 1
        instance variables: hid, the values produced by hidden units
                            y, the values produced by output units
                            hidw, the trainable weight values
                                  [to hidden][from input]
                            w, the trainable weight values
                                  [to output][from hidden]
                            hidbias, the trainable bias values for
                                  hidden units
                            bias, the trainable bias values for
                                  output nodes
        local variables: outputerror, error calculated for each output node
                         hiderror, error calculated for each hidden unit
                         rmse, root-mean-squared-error

        /* The set of inputs are passed into the feedforward function to
           calculate the output for each of the nodes */
        feedforward(x)

        /* allocate space for errors of output nodes and hidden units */
        outputerror <= an array of length same as number of output nodes
        hiderror <= an array of length same as number of hidden units
```

```
/* compute the error of output nodes (explicit target is available
        -- so quite simple) also, calculate the
        root-mean-squared-error to indicate progress */
rmse <= 0
for each output node in the network do
        diff <= desired output - actual output of the node;
        /* calculate output error for weight modification */
        outputerror of the node <= diff *
                outputFunctionDerivative(actual output);
        rmse <= rmse + diff * diff

rmse <= square root of (rmse/number of output nodes in the network)

/* compute the error of the hidden units since these nodes are
        responsible for some fraction of the output error */
for each hidden node in the network do
        sum <= 0.0
        for each output node in the network do
        sum <= sum + weight between current hidden and output node
                * outputerror of the current node
        hiderror of current hidden unit <= sum
                * outputFunctionDerivative(current hidden unit)

        /* update output weights according to errors */
        for each output nodes in the network do
                for each hidden unit in the network do
                 weight between hidden and output nodes <= current outputerror
                                * current hidden unit value * eta
                /* bias can be understood as a weight from a node which is
                        always 1.0. */
                current bias <= current bias
                                + current outputerror * 1.0 * eta

/* change hidden layer weights according to errors */
for each hidden node in the network do
        for each input in inputs do
                weight between input and hidden unit <= the weight +
                        current hiderror * current input * eta
        /* bias can be understood as a weight from a node which is
                always 1.0. */
        current hidbias <= curent hidbias +
                                current hiderror * 1.0 * eta
return rmse
```

Description on Extension of Basic Study with Pseudo Code

In a nutshell, the pseudo code is all about feeding forwards the outputs across the neural network, calculate the error rate and then propagate the error backwards to update the weights and biases. The code of the feedforward function will be shown in the following subsection. For basic understanding, the feedforward function calculates the output values of each of the hidden units based on the inputs, weights and biases and then feeds forwards these values to

the output nodes which essentially perform the same procedures with respective values to produce output values. The error at each node can be calculated by comparing the actual and desired output values. The weights and biases updates are calculated with the formula which involves the already implemented outputFunctionDerivative function.

Implementation of Extension in Java and How to Run

NNClassifier2.java is the classifier class made for the multi-layer neural network machine learning technique. This class is almost identical with the original NNClassifier.java except the instance variable nn is now of type NN2 instead of the already implement NN1. In addition, NNClassifier 2.java has to pass in an additional argument – number of hidden units – when calling the train method of the nn object. Since the change is very little, the code of NNClassifier2.java is not shown here. Please refer to the source file for the implementation.

To use the NNClassifier2 file, simply modify the local variable, c in the TrainClassifier.java file to NNClassifier2 object.

NN2.java is the class which extends the already implemented single-layer neural network which is encapsulated in the class NN1.

NN2.java

```
package machl;

import java.util.Random;

public class NN2 extends NN1 {
	// the values produced by hidden units
	double[] hid;
	// the trainable weight values [to hidden][from input]
	public double[][] hidw;
	// the trainable bias values for hidden units
	public double[] hidbias;

	/** Constructs a multi-layer neural network structure and
	 *  initializes weights
	 *  and bias to small random values.
	 *  @param  nInput  Number of input nodes
	 *  @param  nOutput Number of output nodes
	 *  @param  nHid    Number of hidden nodes
	 *  @param  seed    Seed for the random number generator
	 *                          used for initial weights.
	 */
	public NN2(int nInput, int nOutput, int nHid, int seed) {
		// the super constructor is now responsible for
		//     setting up the input matrix between hidden
		//     layer and output layer
		super(nHid, nOutput, seed);
		// if number of hidden units specified is not valid
		//     set the number to the number of inputs * 2/3
		//       + number of outputs
		if (nHid <= 0)
			nHid = nInput * 2 / 3 + nOutput;
		// allocate space for hidden units and weight values
		//      between input and hidden units
		hid = new double[nHid];
```

```
        hidw = new double[nHid][nInput];
        hidbias = new double[nHid];
        if (rand == null)
              rand=new Random(seed);
        // initialize weight and bias values between hidden
        //     and output layer
        for (int j=0; j<nOutput; j++) {
              for (int i=0; i<nHid; i++)
                    w[j][i]=rand.nextGaussian()*.1;
              bias[j]=rand.nextGaussian()*.1;
        }
}

/** Computes the output values of the output nodes in
 *  the network given input values.
 *  @param  x  The input values.
 *  @return double[] The vector of computed output values
 */
public double[] feedforward(double[] x) {
        // compute the activation of each hidden unit
        //     based on the input value
        for (int j=0; j<hid.length; j++) {
              double sum=0; // reset summed activation value
              for (int i=0; i<x.length; i++)
                    sum+=x[i]*hidw[j][i];
              hid[j]=outputFunction(sum+hidbias[j]);
        }
        // compute the activation of each output unit based
        //    on the hidden layer value
        for (int j=0; j<y.length; j++) {
              double sum=0; // reset summed activation value
              for (int i=0; i<hid.length; i++)
                    sum+=hid[i]*w[j][i];
              y[j]=outputFunction(sum+bias[j]);
        }
        return y;
}

/** Adapts weights in the network given the specification
 *  of which values that should appear at the output
 *  (target) when the input has been presented.
 *  The procedure is known as error backpropagation.
 *  This implementation is "online" rather than
 *  "batched", that is, the change is not based on the
 *  gradient of the global error, merely the local --
 *  pattern-specific -- error.
 *  @param  x  The input values.
 *  @param  d  The desired output values.
 *  @param  eta     The learning rate, always between 0 and
 *                      1, typically a small value, e.g. 0.1
 *  @return double  An error value
 *                      (the root-mean-squared-error).
 */
public double train(double[] x, double[] d, double eta) {

        // present the input and calculate the outputs
        feedforward(x);

        // allocate space for errors of output nodes
        //     and hidden units
        double[] outputerror=new double[y.length];
```

```
            double[] hiderror = new double[hid.length];

            // compute the error of output nodes (explicit target
            //     is available -- so quite simple)
            //     also, calculate the root-mean-squared-error to
            //     indicate progress
            double rmse=0;
            for (int j=0; j<y.length; j++) {
                  double diff=d[j]-y[j];
                  outputerror[j]=diff*outputFunctionDerivative(y[j]);
                  rmse+=diff*diff;
            }
            rmse=Math.sqrt(rmse/y.length);

            // compute the error of the hidden units since these
            //     nodes are responsible for some fraction of the
            //       output error
            for (int j=0; j<hid.length; j++) {
                  double sum = 0.0;
                  for (int i=0; i<y.length; i++) {
                        sum+=w[i][j]*outputerror[i];
                  }
                  hiderror[j]=sum*outputFunctionDerivative(hid[j]);
            }
            // change output weights according to errors
            for (int j=0; j<y.length; j++) {
                  for (int i=0; i<hid.length; i++) {
                        w[j][i]+=outputerror[j]*hid[i]*eta;
                  }
                  // bias can be understood as a weight from a node
                  //     which is always 1.0.
                  bias[j]+=outputerror[j]*1.0*eta;
            }
            // change hidden layer weights according to errors
            for (int j=0; j<hid.length; j++) {
                  for (int i=0; i<x.length; i++) {
                        hidw[j][i]+=hiderror[j]*x[i]*eta;
                  }
                  // bias can be understood as a weight from a node
                  //     which is always 1.0.
                  hidbias[j]+=hiderror[j]*1.0*eta;
            }
            return rmse;
      }
}
```

Performance of Extension on Training and Test Data

Table 3: Implement Hidden Layer and Compare Study: Results

Algorithm	Description	Learning rate	Number of hidden nodes	Number of presentations	Training error	Test error
NN/BP	Single-layer	0.1	Na	50000	243/2803	917/2803
NN/BP	Two-layer	0.1	708	50000	172/2803	837/2803
NN/BP	Two-layer	0.1	1000	50000	211/2803	871/2803
NN/BP	Two-layer	0.1	500	50000	179/2803	838/2803
NN/BP	Two-layer	0.1	100	50000	173/2803	825/2803
NN/BP	Two-layer	0.1	80	50000	123/2803	727/2803
NN/BP	Two-layer	0.1	70	50000	127/2803	723/2803
NN/BP	Two-layer	0.1	50	50000	123/2803	719/2803
NN/BP	Two-layer	0.1	40	50000	119/2803	713/2803
NN/BP	Two-layer	0.1	30	50000	130/2803	722/2803
NN/BP	Two-layer	0.1	20	50000	185/2803	819/2803
NN/BP	Two-layer	0.1	10	50000	2371/2803	2431/2803

The original dataset was divided into half – the first half was the training set; the second half was the test set. The learning rate and number of presentations were fixed throughout the assessment to observe the effect of changes in the number of hidden units upon the test performance.

The first experiment used the already implemented single-layer neural network and it has 917 errors out of the 2803 test sets. From the second case onwards, the implemented two-layer (one hidden layer) neural network was used and both training and test performance had improved. The result shows that when number of hidden units was in the range of 30 to 80, the classifier had the best overall training and test performance. The interesting observation is although low number of hidden units were favourable, too few hidden units for example, 10 in the last result, would lead to horrible performance reduction.

Conclusion of the Study

In summary, multi-layer neural network generally has greater performance than the single-layer neural network in classifying the handwritten characters. Nevertheless, the number of

hidden units chosen can greatly affect the test performance. If the number of hidden units are too few or too many, it might lead to unfavourable result as what was shown in the investigation.

2B. ID3 Decision Tree with Pruning

Aims of the Study

Extends the original decision tree ID3 to implement extra feature, which is decision tree pruning. A decision tree with pruning is implemented to avoid overfitting problem, hence it often builds a smaller tree in size, make it is easier to be understood.

Description on the Extension of the Basic Implementation

The existing decision tree was implemented using binary tree structure. The machine learning algorithm was implemented by taking training set and partition according to the attribute, which is specific pixel in the bitmap. In addition, the attribute with greatest information gain in the attributes subset will be split recursively until there is no more partition data subset to be mapped. Therefore, the leaf node is always a node with class value.

The ID3 pruning implementation is basically similar to the algorithm stated above, but before splitting the attribute to subtrees, it checks whether the chi-square distribution of current selected attribute on partition data subset is significant at the 5% level, if it is, the attribute will be split and continue, otherwise, algorithm will reselect attribute with next greatest information gain, the step is repeated until the attribute selected is significant at the 5% level.

The implementation of the extension was done with reference to Russel and Stuart's *Artificial Intelligence, 3rd edition* (2009). As well, extra classes are included in the implementation to calculate Chi-square probability 1-Q(1,d). The extra classes are provided by third party software site Colt. The Colt software packages are copyrighted and shown below:

Packages cern.colt* , cern.jet*, cern.clhep

Copyright (c) 1999 CERN - European Organization for Nuclear Research.

Pseudo Code of the Implementation

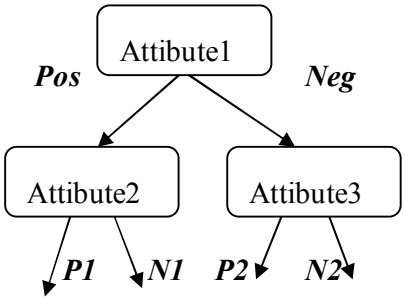
```
Function DECISION-TREE-LEARNING(examples, attributes, default) returns a decisiontree
        Inputs:  examples, set of examples
                  Attributes, set of attributes
                  Default, default value for the goal predicate
        If examples is empty then
                return default
        Else if all examples have the same classification then
                return the classification
        Else if attributes is empty then
                return MAJORITY-VALUE(example)
        Else
                While TRUE:
                        Best = CHOOSE-ATTRIBUTE(attributes, examples)
                        If CHI-SQUARE-TEST(attribute, examples) <= 5% then
                                Break;
                        Attributes = attributes excludes best
                End
```

```
        Tree = a new decision tree with root test best
        For each value v_i of best do
                Examples_i = {elements of examples with best = v_i}
                Subtree = DECISION-TREE-LEARNING(examples, attributes – best,
                                                 MAJORITY-VALUE(examples))
                Add a branch to tree with label v_i and subtree subtree
        End
        Return tree

Function CHI-SQUARE-TEST(attribute, examples) return  probability
    Input:   attribute, selected attribute
             examples, set of examples
    pos = | Examples that satisfy the attribute|
    neg =  | Examples that satisfy the attribute|
    leftExpectedPos = pos * (p1+n1)/ |examples|
    leftExpectedNeg = neg * (p1+n1)/ |Examples|
    rightExpectedPos = pos * (p2+n2)/ |examples|
    rightExpectedNeg = neg * (p2+n2)/ |Examples|
    (p_i, n_i is the |subset of examples| that satisfy and not satisfy the subtree-attribute)
    Distribution = (pos-leftExpectedPos)^2/leftExpectedPos +
                   (neg-leftExpectedNeg)^2/leftExpectedNeg +
                   (pos-rightExpectedPos)^2/rightExpectedPos +
                   (neg-rightExpectedNeg)^2/rightExpectedNeg
    Return 1-Q(1,distribution)
```

Description on Extension of Basic Study with Pseudo Code

In the DECISION-TREE-LEARNING function, an infinite loop is added before splitting attribute into subtrees. Inside the while loop, it does a chi-square-test on the attribute according to partitioned set of examples. A chi-square-test operates as following, lets *pos* be the number of examples that satisfy the attribute, and *neg* be the number of examples that is not satisfy the attribute. By further splitting the example subset into p_i and n_i, as the figure below.



Then, *pos, neg, p1, n1, p2, n2* will substitute into the formula to calculate $expectedPositive_i$ and $expectedNegative_i$, since this is a binary tree, i value is either left or right. Then, the $expectedPositive_i$ and $expectedNegative_i$ are used to calculate chi-square distribution. Lastly, distribution value will be used to calculate probability. In this case, 1 is refers to the degree of freedom (Value of attribute - 1). Because it is binary tree, there are only two value of each attribute, thus, degreeOfFreedom is equals to 1.

Implementation of Extension in Java and How to Run

PruneID3Classifier.java is the classifier class made for the Decision tree pruning machine learning technique. This class extended ID3Classifier and is almost identical with the original ID3Classifier, besides that it creates a PruneBinID3 instead of BinID3.

To use the classifier, simply modify the local variable, c in the TrainClassifier.java file to PruneID3Classifier object.

Moreover, PruneBinID3.java is the class which extends the already implemented BinID3 and includes the decision-tree-learning method which had described in the pseudo code section.

PruneID3Classifier.java

```
/**
 * Packages cern.colt* , cern.jet*, cern.clhep
 *
 * Copyright (c) 1999 CERN - European Organization for Nuclear Research.
 */

package bitmap;

import machl.*;

public class PruneID3Classifier extends ID3Classifier {
	private static String name = "Prune ID3 Classifier";
	private PruneBinID3 pID3 = null;

	public PruneID3Classifier(int nRow, int nCol) {
		super(nRow, nCol);
	}

	  /**
	   * Trains the ID3 classifier on provided samples.
	   * @param maps the bitmaps which are used as training inputs
	   */
	  public void train(ClassifiedBitmap[] maps) {
	    features=new boolean[maps.length][];
	    targetValues=new String[maps.length];
	    for (int p=0; p<maps.length; p++) {
	      features[p]=((Bitmap)maps[p]).toBooleanArray();
	      features[p] = centralize(features[p]);
	      targetValues[p]=getLabel(maps[p].getTarget());
	    }
	    pID3 = new machl.PruneBinID3(labels, features, targetValues,
	            classValues);
	    tree = pID3.induce();
	  }

	public String getName(){
		return name;
	}
}
```

PruneBinID3.java

```
/**
 * Packages cern.colt* , cern.jet*, cern.clhep
 *
 * Copyright (c) 1999 CERN - European Organization for Nuclear Research.
 */
package machl;

import cern.jet.stat.Probability;

public class PruneBinID3 extends BinID3{

     public PruneBinID3(String[] label, boolean[][] input, String[] output,
                        String[] classes){
           super(label, input, output, classes);
      }

      public BinTree induce() {
          // the initial partition contains all the samples available to us
          int[] entries=new int[input.length];
          for (int i=0; i<entries.length; i++)
            entries[i]=i;
          // the initial features include all available
          int[] features=new int[label.length];
          for (int i=0; i<features.length; i++)
            features[i]=i;
          // ok, let's get the show on the road
          return induceTree(entries, features);
      }

      public BinTree induceTree(int[] partition, int[] features) {
          // if the partition is empty, we can not return a tree
          if (partition.length==0) {
            return null;
          }
          // check if all entries in partition belong to the same class. If
            so, return node, labeled with class value
          int[] classCnt=new int[classes.length];
          String sameValue=null;
          boolean sameClass=true;
          for (int p=0; p<partition.length; p++) {
            String targetValue=output[partition[p]];
            for (int n=0; n<classes.length; n++) {
              if (targetValue.equals(classes[n])) {
                classCnt[n]++;
                break;
              }
            }
            if (p==0)
              sameValue=targetValue;
            else {
              if (!sameValue.equalsIgnoreCase(targetValue)) {
                sameClass=false;
                break;
              }
            }
          }
          if (sameClass)  {
            return new BinTree(sameValue);
          } else {
```

```
  int max=0;
  for (int n=1; n<classes.length; n++)
    if (classCnt[max]<classCnt[n])
      max=n;
  if ((double)classCnt[max]/(double)partition.length>0.50 ||
        partition.length<5) {
        // if more than 50% of samples in partition are of the
        same class OR fewer than 5 samples
    System.out.print(".");
    return new BinTree(classes[max]);
  }
}

// if no features are available, we can not return a tree
if (features != null && features.length==0) {
  return null;
}

// class values are not equal so we select a particular feature
to split the partition
int selectedFeature=selectFeature(partition, features);

// create new partition of samples
// use only corresponding subset of full partition
int[] partTrue=matches(partition, selectedFeature, true);
int[] partFalse=matches(partition, selectedFeature, false);
// remove the feature from the new set (to be sent to subtrees)

int[] nextFeatures = features;
  while (true){

        nextFeatures = new int[nextFeatures.length-1];
      int cnt=0;
      for (int f=0; f<features.length; f++) {
        if (features[f] != selectedFeature)
          nextFeatures[cnt++]= features[f];
      }

      if (nextFeatures != null && nextFeatures.length==0) {
          return null;
      }
        int pos = partTrue.length;
        int neg = partFalse.length;

        int leftFeature = selectFeature(partTrue, nextFeatures);
       int rightFeature = selectFeature(partFalse, nextFeatures);

       int leftPos = matches(partTrue, leftFeature, true).length;
        int leftNeg = matches(partTrue, leftFeature,
                              false).length;
        int rightPos = matches(partFalse,
                              rightFeature,true).length;
        int rightNeg = matches(partFalse,
                              rightFeature,false).length;

        double leftExpectedPos = pos * (leftPos + leftNeg)/(pos +
                              neg);
        double leftExpectedNeg = neg * (leftPos + leftNeg)/(pos +
                              neg);
        double rightExpectedPos = pos * (rightPos +
                              rightNeg)/(pos + neg);
```

```
            double rightExpectedNeg = neg * (rightPos +
                                    rightNeg)/(pos + neg);

            double delta = 0;
            if (leftExpectedPos > 0)
                  delta += Math.pow((leftPos -
                        leftExpectedPos),2)/leftExpectedPos;
            if (leftExpectedNeg > 0)
                  delta += Math.pow((leftNeg -
                        leftExpectedNeg),2)/leftExpectedNeg;
            if (rightExpectedPos > 0)
                  delta += Math.pow((rightPos -
                        rightExpectedPos),2)/rightExpectedPos;
            if(rightExpectedNeg > 0)
                  delta += Math.pow((rightNeg -
                        rightExpectedNeg),2)/rightExpectedNeg;
            if (Probability.chiSquareComplemented(1, delta) <= 0.05)
                  break;

            selectedFeature = selectFeature(partition, nextFeatures);
            partTrue = matches(partition, selectedFeature, true);
            partFalse = matches(partition, selectedFeature, false);
            features = nextFeatures;
      }
     // construct the subtrees using the new partitions and reduced
     set of features
     BinTree branchTrue=induceTree(partTrue, nextFeatures);
     BinTree branchFalse=induceTree(partFalse, nextFeatures);

    // if either of the subtrees failed, we have confronted a problem,
    use the most likely class value of the current partition
     BinTree defaultTree=null;
     if (branchTrue==null || branchFalse==null) {
       // indicate a majority vote
       int[] freq=new int[classes.length];
       int most=0;
       for (int c=0; c<classes.length; c++) {
         int[] pos = matches(partition, classes[c]);
         freq[c] = pos.length;
         if (freq[c]>=freq[most])
           most=c;
       }
       // the majority class value can replace any null trees...
       defaultTree=new BinTree(classes[most]);
       if (branchTrue==null && branchFalse==null)
         return defaultTree;
       else // return the unlabeled node with subtrees attached
         return new BinTree(labelFeature(selectedFeature),
                   (branchTrue==null?defaultTree:branchTrue),
                   (branchFalse==null?defaultTree:branchFalse));
     } else {
       // if both subtrees were successfully created we can either
       if (branchTrue.classValue != null && branchFalse.classValue !=
             null) {
         if (branchTrue.classValue.equals(branchFalse.classValue)) {
           // return the the current node with the classlabel common
             to both subtrees, or
           return new BinTree(branchTrue.classValue);
         }
       }
       // return the unlabeled node with subtrees attached
```

```
            return new BinTree(labelFeature(selectedFeature), branchTrue,
                  branchFalse);
          }
        }
}
```

Performance of Extension on Training and Test Data

Table 4: Basic Study: ID3 Decision Tree with Pruning Optimisation

Algorithm	Description	Training error	Test error
ID3 Pruning	Binary tree with pruning	153/1000	593/1000
ID3 Pruning	Binary tree with pruning	309/2000	1361/2000
			198/1000
			111/1000
			2623/4606
ID3 Pruning	Binary tree with pruning	1032/4606	666/2803
			917/3606
			479/2000
			908/4000
			364/1000

Conclusion of the Study

In conclusion, a chi-square pruned decision tree performs less accurately than a normal decision tree. However, they possess same behaviour such that they trained by 75% data set will perform better to a 25% test data set.

2C. Different Machine Learning Technique

Aims of the Study

Besides neural network and decision tree machine learning techniques, there are different machine learning techniques discovered by other people. Naive Bayes classifier is one of them and is chosen here to implement. The implementation aims to study how a Naive Bayes classifier will predict a character after trained by certain number of training data set.

Description on the Extension of the Basic Implementation

Amongst these classifiers, Naive Bayes Classifier is one of the easiest to implement. By input training set to the Naive Bayes classifier, it will be trained by recoding total number of each of all classes, denoted count(Class), and in each class, count and record the total number of value of each feature, denoted count(feature[i] ^ C). After classifier being trained, the classifier will predict the test result by counting probability of all classes, denoted P(C), and only return the class with highest probability. The implementation will be explained in details in the sections below.

Pseudo Code of the Implementation

```
Constructor NB( features, targetValues, numOfClasses)
        Input: features, a set of examples'with thier features
                targetValues, training output set
                numOfClasses, number of target classes
        features = features;
        targetValues = targetValues
        numOfClasses= numOfClasses


Function train()
        For i = 0 to |targetValues|
                Index = index of a targetValue
                featuresOfChar[index][0]++ // increment the number of this class
                for f = 0 to |features[i]|
                        if (features[i][f] == true) then
                                featuresOfChar[index][f+1]++
                                //featuresOfChar[index][0] reserved for storing total
                                number of such class
                End
        End

Function getProbability(feature) returns an array of probabilities
        Input: feature, a set of features of a data that wanted to examine
        Probabilities[numOfClasses]
        For i = 0 to numOfClasses
                Probability[i] = count(class i)/ count(all)
                For f = 0 to |feature|
                        If (feature[f] == true) then
                                Probability[i] = probability[i]
                                        * count(feature[f] == true ^ class i)/ count(class i)
                        Else then
                                Probability[i] = probability[i]
                                        * count(feature[f] == false ^ class i)/ count(class i)
                End
        End
        Return Probability
```

Description on Extension of Basic Study with Pseudo Code

Basically, there are two methods and one constructor required in order to implement Naive bayes classifier. The constructor is used to initialize the values of the variables, which can be used within the classes.

In addition, the two methods are train and getProbability. The train method tends to learn and record the count all the feature[i][x] of in their corresponding class. The class array is a two-dimensional array such that, array[class index][0] always holds the total number of the class appeared in dataset, array[class index][1] to array[class index][|feature|] holds the total true count of feature[i] (consider only a feature has either true or false value).

On the other hand, the getProbability will be invoked while testing a data. Firstly, it initializes an array called probability of length equals to |target classes| that used to store the probability of each class. For each class, it will multiply probability of a class, which equals to (total number of target class / total number of data set trained) with probability of each feature[i] presented in test data (count(feature[i] == (true/false) ^ class)/ count(class)). After probability of all target classes are calculated, the array will be returned. Lastly, Naive Bayes classifier will predict the class with highest probability.

The implementation of the extension was done with reference to Russel and Stuart's *Artificial Intelligence, 3rd edition* (2009).

Implementation of Extension in Java and How to Run

NBClassifier.java is the classifier class made for the Naive Bayes model machine learning technique. This classifier has not includes constructor, so default constructor will be called when this object is instantiated.

In order to use the classifier, simply modify the local variable, c in the TrainClassifier.java file to NBClassifier object.

In addition, NB.java is the class includes in machl package, which indicate machine learning is operate in this class. For implementation detail, please refers to pseudo code section.

NBClassifier.java

```
package bitmap;

import machl.*;

public class NBClassifier extends LetterClassifier{
	private static String name = "Naive Bayes Classifier";

	/** Naive Bayes learning*/
	private NB nb = null;
	/** training output */
	private String[] targetValues = null;
	/** the 32 * 32 features of n number of letter*/
	private boolean[][] features = null;

	/**
	 * Test the test data, and return the probabilities of all possible
	 * results
	 * @param map     test set
```

```
         * @return        probabilities of all possible results
         */
        public double[] test(Bitmap map) {
              boolean[] feature = centralize(map.toBooleanArray());
              return nb.getAllProbabilities(feature);
        }

        /**
         * Train the machine with given training set.
         * @param maps    training set
         */
        public void train(ClassifiedBitmap[] maps) {
              features = new boolean[maps.length][];
            targetValues = new String[maps.length];
            for (int p=0; p<maps.length; p++) {
              features[p]=((Bitmap)maps[p]).toBooleanArray();

              features[p] = centralize(features[p]);
              targetValues[p]=getLabel(maps[p].getTarget());
            }
            nb = new NB(features, targetValues, getClassCount());
            nb.train();
        }

        /**
         * Get the name of this classifier
         * @return name of the classifier
         */
        public String getName() {
              return name;
        }

}
```

NB.java

```
package machl;

import java.util.*;
import java.io.*;

public class NB implements Serializable{
        /** training output */
        private String[] targetValues;
        /** the 32 * 32 features of n number of letter*/
        private boolean[][] features;
        /** the total Features count for each letter A-Z*/
        private int[][] featuresOfChar;

        /**
         * A constructor that initialize all attributes.
         *
         * @param features      the 32 * 32 features of n number of letter
         * @param targetValues  training output
         * @param numOfClasses  total target classes
         */
        public NB(boolean[][] features, String[] targetValues, int
                        numOfClasses) {
              this.targetValues = targetValues;
              this.featuresOfChar = new
                                  int[numOfClasses][features[0].length + 1];
```

```
            this.features = features;
        }

        /**
         * Return the probabilities of all the classes, which used to predict
         * the result of a test data based on learned dataset.
         *
         * @param feature 32*32 features of a letter
         * @return              probabilities of all classes
         */
        public double[] getAllProbabilities(boolean[] feature){
             double[] probabilities = new double[this.featuresOfChar.length];
              //Given X is the features that wanted to examine
              //Calculate probabilities of all classes
              for(int i = 0; i < featuresOfChar.length; i++){
                    // P(C) = count(C) / count(all)
                    probabilities[i] = (double)(featuresOfChar[i][0]) /
                                targetValues.length;
                    for(int f = 0; f < feature.length; f++){
                          if(feature[f])
                                //P(pixel = true | X = true) =  count(pixel =
                                      true ^ C) / count(C)
                                probabilities[i] *=
                                      (double)(featuresOfChar[i][f+1]) /
                                            featuresOfChar[i][0];
                          else
                                //P(pixel = false | X = false) =  count(pixel
                                      = false ^ C) / count(C)
                                probabilities[i] *=
                                      (double)(featuresOfChar[i][0]-
                                            featuresOfChar[i][f+1]) /
                                            featuresOfChar[i][0];
                    }
              }
              return probabilities;
        }
        /**
         * The machine learning method that learn based on training set,
         * which is targetValues in this case.
         */
        public void train() {
              for (int i = 0; i < targetValues.length; i++) {
                    int index = targetValues[i].charAt(0) - 'A';
                    // increment the character's count
                    featuresOfChar[index][0]++;
                    for (int f = 0; f < features[i].length; f++){
                          if(features[i][f])
                                //let (f+1) = r * 32 + c, increment if true
                                value is found
                                featuresOfChar[index][f+1]++;
                    }
              }
        }

}
```

Performance of Extension on Training and Test Data

Table 5: Basic Study: Naive Bayes clasifier Optimisation

Algorithm	Description	Training error	Test error
NB	Naive Bayes classifier	1582/4606	1387/4000
			433/2000
			193/1000
			240/1000
NB	Naive Bayes classifier	312/2000	957/2000
			185/1000
			1809/3606
NB	Naive Bayes classifier	71/1000	485/1000
			1186/2000
NB	Naive Bayes classifier	592/2803	137/1000

Conclusion of the Study

In conclusion, although Naive Bayes model possess a very simple but reasonably powerful algorithm, but it generally performs slightly less accurate on test set than a decision tree does. After observed the performance, the optimum performance of the Naive Bayes classifier is when it was trained by a reasonably number of training data, which is 2803 data set in this case. Otherwise, either too little or large training set may leads Naive Bayes classifier to either overtrained or lack of training.

3A. Create an Ensemble of Classifiers

Aims of the Study

To create a new classifier that is an ensemble of multiple classifiers implemented in the previous studies.

To optimise and assess the performance of the classifier ensemble.

Description on Classifier Ensemble

Ensemble learning combines the predictions of several individually trained classifiers into a single composite classification which aims to mitigate the error of any weak learning algorithm (Pal, 2007). The combination of predictions was done by using a specified majority voting which made the misclassification less likely (Russel & Norvig, 2009).

Description on Implementation of Classifier Ensemble

The ensemble learning adopted is called AdaBoost (Adaptive Boosting). It is a variant of the widely-used boosting method for ensemble learning. The algorithm generates a set of hypothesis by weighting the training examples (Russel & Norvig, 2009). Ultimately the algorithm produces an overall prediction in response to a test pattern by combining the results or predictions from multiple classifiers via a so-called "weight-majority" voting system.

Pseudo code for AdaBoost from the textbook (Russel & Norvig, 2009):

```
function ADABOOST (example, L, K) returns a weighted-majority hypothesis
        inputs: examples, set of N labelled examples (X_i, y_i), . . . , (X_N, y_N)
                L, a learning algorithm
                M, the number of hypotheses in the ensemble
        local variables: w, a vector of N example weights, initially 1./N
                         h, a vector of M hypotheses
                         z, a vector of M hypothesis weights
        for k = 1 to K do
                h[k] = L(examples, w)
                error = 0
        for j=1 to N do
                if h[k] (x_j) ≠ y_j then error = error + w[j]
        for j=1 to N do
                if h[k] (x_j) = y_j then w[j]= w[j] . error / ( 1 – error)
        w = NORMALIZE(W)
        z[k]= log (1 - error) / error

        return WEIGHTED-MAJORITY(h, z)
```

The above pseudo code was modified and adapted for the classifier. The modification was made as such.

```
function train returns void
        input: maps: the bitmaps which are used as training inputs
        instance variables: cList: a list of all classifiers used
                classifierWeights: map storing classifier-weight pairs
        initialiseBitmapWeights(maps.length)
        initialiseClassifierWeights()
        for each c in cList do
            if (c instanceof ID3Classifier)
                ((ID3Classifier)cList[0]).train(maps)
            else if (c instanceof NNClassifier) {
                ((NNClassifier)cList[1]).train(maps, 50000, 0.1)
            else if (c instanceof NNClassifier2)
                ((NNClassifier2)cList[2]).train(maps, 50000, 0.1)
            else if (c instanceof NBClassifier)
                ((NBClassifier)cList[3]).train(maps)
            else
                throw exception
            error = calcError(maps, c)
            if (error < 0.0001)
                break the loop
            updateBitmapWeights(maps, c, error)
            newClassifierWeight = classifierWeights.get(c)
                * Math.log((1.0 - error) / error)
            classifierWeights.put(c, newClassifierWeight)
```

The above pseudo code describes the program flow of the train method. Basically each of the classifiers are trained with the dataset given. In each round, the error will be calculated and will be used to update the weight for each example and also the weight for the classifiers.

The test process is similar to the AdaBoost pseudo code as shown at the beginning of this subsection. Please refer to the source code for the actual implementation.

Implementation in Java and How to Run

The implementation was done with the reference to Ravi Mohan(2010)'s source code provided on the Internet. To use the ClassifierEnsemble class, simply modify the local variable, c in the TrainClassifier.java file to ClassifierEnsemble object. To manipulate the classifiers for ensemble, simply modify the initialisation value of the cList to the correct number of classifiers and then initialise the each of the elements to the correct class respectively. This class assumes that there will be only four types of algorithms for ensemble, namely ID3Classifier, NNClassifier, NNClassifier2 and NBClassifier. ID3Classifier with pruning is not included since it would probably worsen the performance.

ClassifierEnsemble.java

```
package bitmap;

import java.util.Hashtable;

/**
 * This class is an ensemble of multiple classifiers of different
 * machine learning techniques for letter recognition.
 *
 * @author SU Sheng Loong & TEE Lip Jian
 */
```

```
public class ClassifierEnsemble extends LetterClassifier {
	private static String name="Classifier Ensemble";
	private LetterClassifier[] cList;
	private double[] bitmapWeights;
	private Hashtable<LetterClassifier, Double> classifierWeights;

	/**
	   * Construct a classifier ensemble for bitmaps of specified size.
	   *
	   * @param nRows number of rows in the bitmap
	   * @param nCols number of columns in the bitmap
	   */
	public ClassifierEnsemble(int nRows, int nCols) {
		setClassifierList(new LetterClassifier[4]);
		cList[0] = new ID3Classifier(nRows, nCols);
		cList[1] = new NNClassifier(nRows, nCols);
		cList[2] = new NNClassifier2(nRows, nCols);
		cList[3] = new NBClassifier();
	}

	/**
	 * Identifies the classifier, e.g. by the name of the
	 * author/contender
	 * @return the identifier
	 */
	public String getName() {
		return name;
	}

	/**
	 * Getter method for the list of all classifiers embodied
	 * @return the classifier list
	 */
	public LetterClassifier[] getClassifierList() {
		return cList;
	}

	/**
	 * Setter method for the list of all classifiers embodied
	 * @param cList a list of LetterClassifier objects
	 */
	public void setClassifierList(LetterClassifier[] cList) {
		this.cList = cList;
	}

	/**
	 * Classifies the bitmap
	 * @param map the bitmap to classify
	 * @return the probabilities of all the classes (should add up to 1).
	 */
	public double[] test(Bitmap map) {
		return weightedMajority(map);
	}

	/**
	 * Trains the ClassifierEmsemble classifier on provided samples.
	 * @param maps the bitmaps which are used as training inputs
	 */
	public void train(ClassifiedBitmap[] maps) {
		initialiseBitmapWeights(maps.length);
		initialiseClassifierWeights();
```

```
		for (LetterClassifier c: cList) {
			if (c instanceof ID3Classifier) {
				((ID3Classifier)cList[0]).train(maps);
			}
			else if (c instanceof NNClassifier) {
				((NNClassifier)cList[1]).train(maps, 50000, 0.1);
			}
			else if (c instanceof NNClassifier2) {
				((NNClassifier2)cList[2]).train(maps, 50000, 0.1);
			}
			else if (c instanceof NBClassifier) {
				((NBClassifier)cList[3]).train(maps);
			}
			else
				throw new RuntimeException(
				"Classfier type not supported!");
			double error = calcError(maps, c);
			if (error < 0.0001)
				break;
			updateBitmapWeights(maps, c, error);
			double newClassifierWeight = classifierWeights.get(c)
				* Math.log((1.0 - error) / error);
			classifierWeights.put(c, newClassifierWeight);
		}
		System.out.println("Train completed");
	}

	/**
	 * Initialise all bitmap weights
	 * @param size number of bitmaps for training
	 */
	public void initialiseBitmapWeights(int size) {
		double value = 1.0 / (1.0 * size);
		bitmapWeights = new double[size];
		for (int i = 0; i < size; i++)
			bitmapWeights[i] = value;
	}

	/**
	 * Initialise all classifier weights
	 */
	public void initialiseClassifierWeights() {
		classifierWeights = new Hashtable<LetterClassifier, Double>();
		for (LetterClassifier c: cList)
			classifierWeights.put(c, 1.0);
	}

	/**
	 * Calculate error made by the classifier when trained on
	 * the bitmaps.
	 *
	 * @param maps dataset for training
	 * @param c the letter classifier to be trained
	 * @return the error calculated
	 */
	public double calcError(ClassifiedBitmap[] maps, LetterClassifier c)
	{
		double error = 0.0;
		for (int i = 0; i < maps.length; i++) {
			int actual=c.index((Bitmap)maps[i]);
			int target=maps[i].getTarget();
```

```
            // if actual output is not same as desired output,
            //          increment error by the example weight
            if (actual != target)
                  error = error + bitmapWeights[i] * 1;
      }
      return error;
}

/**
 * Update the bitmap weights after training based
 * on the error calculated
 *
 * @param maps the bitmaps for training
 * @param c the trained classifier
 * @param error the error calculated during training
 */
private void updateBitmapWeights(ClassifiedBitmap[] maps,
            LetterClassifier c, double error) {
      double epsilon = error / (1.0 - error);
      for (int i = 0; i < maps.length; i++) {
            int actual=c.index((Bitmap)maps[i]);
            int target=maps[i].getTarget();
            // if actual output is not same as desired output,
            //          increment weight by epsilon
            if (actual != target)
                  bitmapWeights[i] = bitmapWeights[i] * epsilon;
      }
      bitmapWeights = normalise(bitmapWeights);
}

/**
 * Normalise the value in an array of double variables.
 *
 * @param weights an array of weights
 * @return the normalised array of weights
 */
public double[] normalise(double[] weights) {
      double total = 0.0;
      for (double w : weights)
            total = total + w;
      double[] normalised = new double[weights.length];
      if (total != 0)
            for (int i = 0; i < weights.length; i++)
                  normalised[i] = weights[i] / total;
      double totalN = 0.0;
      for (double d : normalised)
            totalN = totalN + d;
      return normalised;
}

/**
 * Calculate the probability for each of the classifications
 * by using the weighted majority algorithm.
 *
 * @param map the bitmap for testing
 * @return the probability for each of the classifications
 */
public double[] weightedMajority(Bitmap map) {
      double[][] table=new double[cList.length][getClassCount()];
      for (int i=0; i<cList.length; i++) {
            int predicted=cList[i].index(map);
```

```
                        table[i][predicted] += classifierWeights.get(cList[i]);
                }
                double[] out = new double[getClassCount()];
                for (int i=0; i<getClassCount(); i++)
                        out[i] = scoreOfClass(i, table);
                return out;
        }

        /**
         * Calculate the score for each classification.
         *
         * @param classification a particular classification
         * @param table holds the score for each classification
         * @return score of the classification
         */
        public double scoreOfClass(int classification, double[][] table) {
                double score = 0.0;
                for (int i = 0; i < table.length; i++)
                        score += table[i][classification];
                return score;
        }

}
```

Performance of Extension on Training and Test Data

Table 6: Implement Classifier Ensemble: Results

Algorithm	Description	Learning rate	Number of hidden nodes	Number of presentations	Training error	Test error
NN/BP	Single-layer	0.1	Na	50000	246/2803	961/2803
NN/BP	Multi-layer	0.1	30	50000	134/2803	740/2803
ID3 DT	Binary tree without pruning	Na	Na	Na	313/2803	1653/2803
Ensemble	ID3+NN+NN2	0.1	30	50000	109/2803	756/2803
Ensemble	ID3+NN+NN2	0.1	30	50000	192/2803	494/2803
Ensemble	ID3+NN+NN2+NB	0.1	30	50000	120/2803	737/2803
Ensemble	ID3+NN+NN2+NB	0.1	30	50000	207/2803	507/2803

The table above showcase the results obtained from training and testing the classifier ensemble. For the first two Ensemble cases embodied three algorithms – ID3 decision tree without pruning, single-layer neural network and multi-layer neural network. The last two cases embodied an additional algorithm – Naïve Bayes algorithm which was implemented in the previous study. ID3 decision tree with pruning is not chosen because it generally results in worse performance.

All tests in the table show that the ensemble did have improvement in the accuracy of the letter recognition over the previous studies. When different training and test sets were used, the performance varied by the range of 200 to 250. Possible reasons were the noise or due to overtraining. Nevertheless, the overall performance of the ensemble is decent.

Note that the learning rate, number of hidden nodes and number of presentations were fixed at the numbers which were deemed to be optimal based on the previous studies.

Conclusion of the Study

Ensemble learning is a good optimisation method to reduce the chances of making wrong hypothesis since multiple of classifiers are being used for classifying the letters. Problems such as noise and overtraining can be reduced and thus produce more accurate predictions.

3B. Implement a pre-processing technique

Aims of the Study

Pre-processing technique is one of optimisations that could be implemented to let the classifiers obtaining more accurate prediction on test set.

Description on pre-processing technique

The pre-processing technique chosen here is to center the written character. By default, user may write the character in the writing panel bias to either left or right side. It results inconsistency between trained and test data such that trained data may bias to left side and validates against a character bias to right side, although they are written same character, the classifier may more likely to predict another different character which is bias same side with the test data. Hence, by implementing this technique, it centring all the characters and ensure the validation is consistently done, which only focus on validating center area.

Description on Implementation of pre-processing

A method called centralize is included in the parent-class LetterClassifier which is being extended by other classifiers. Within this method, it will first find out the starting column and end column of the written character, then calculate the character's center axis. After that, it determines whether the character should shift left or right. There are certain conditions we needed to know. Firstly, the center axis of a 32* 32 pixels image is the column 15 (consider columns from 0 to 31). Secondly, there are three variables charStart, charEnd, charCenter, which holds the column number of starting, end and center of a character.

If it is a shift left operation, it moves the character to the center by following these steps. It will start cutting the charStart column to (15 – (charCenter – charStart)) column, and this replacement will continue to next column, until it reaches the charEnd column.

Otherwise, it will be a shift right operation, which worked oppositely to the shift right operation, started to cut charEnd column to (15 + (charEnd – charCenter)) column, and then move on, until it reaches the charStart column.

Implementation in Java and How to Run

As mentioned above, this method is included in LetterClassifier.java. In order to use this method, it is needed to be called every time before a feature passed into train or test method in the machine learning classes.

```
public boolean[] centralize(boolean[] feature){
        int charStart = 31, charEnd = 0;

        for (int r = 0; r < 32; r++) {
              boolean end = true;
              for(int c = 0; c < 32; c++) {
                    if (feature[r*32+c]) {
                          if(r <= charStart)
                                charStart = r;
                          end = false;
                    }
              }
              if(end)
                    charEnd = r;
        }
        int charCenter = (charStart + charEnd) / 2;
        //shift left
```

```
        if (charCenter > 15){
              for(int x = (15 - (charCenter - charStart));
                                charStart < charEnd; x++, charStart++) {
                    for( int c = 0; c < 32; c++){
                          feature[x * 32 + c] = feature[charStart * 32 + c];
                           if (feature[x * 32 + c] !=
                                              feature[charStart * 32 + c])
                                 feature[charStart * 32 + c] = false;
                    }
              }
        } else { //shift right
              for(int x = (15 + (charEnd - charCenter)); charEnd >
                                              charStart; x--, charEnd--) {
                    for( int c = 0; c < 32; c++){
                           feature[x * 32 + c] = feature[charEnd * 32 + c];
                           if (feature[x * 32 + c] !=
                                              feature[charEnd * 32 + c])
                                 feature[charEnd * 32 + c] = false;
                    }
              }
        }
        return feature;
  }
```

Performance of Extension on Training and Test Data

Table 7: Basic Study: Naive Bayes classifier with pre-processing Optimisation

Algorithm	Description	Training error	Test error
NB	Naive Bayes classifier	1453/4606	1281/4000
			379/2000
			173/1000
			206/1000
NB	Naive Bayes classifier	227/2000	892/2000
			97/1000
			1694/3606
NB	Naive Bayes classifier	37/1000	484/1000
			1155/2000
NB	Naive Bayes classifier	540/2803	119/1000

Conclusion of the Study

The testing is done on Naive Bayes classifier, by comparing the results before and after implementing pre-processing technique, Naive Bayes classifier with pre-processing have less training and test error than the Naive Bayes classifier before implement pre-processing.

3C. Overtraining prevention

Aims of the Study

To study how overtraining of classifier can be prevented by using validation set.

To implement validation and assess performance of the classifier after the implementation.

Description on Overtraining Prevention

In order to prevent overtraining of the classifier, validation is used when training the classifier. The technique makes use of an independent validation set in addition to the training set. The datasets initially for training will be divided into the new training and validation set.

Description on Implementation of Cross-validation

```
divide training set into training set and validation set
trainingError = 0.0
validationError = 0.0
for each iteration do
        sample = a random integer
        train(trainingsets[sample])
        test with the validation set
        calculate error rate for test
        if (validationError same as trainingError)
                early stopping
```

The above simplified pseudo code describes how the validation is done throughout the training. As mentioned before, the initial training set is divided into the new training set and validation set. The method will then go through the iterations to train on the training set and test on the validation sets based on the random value. If the error rate of validation set and training set converges, the training will stop.

Implementation in Java and How to Run

As usual, change the classifier object in TrainClassifier.java to the object of type NN2ClassifierValidation. When the method train is called, the parameters should be passed in correctly based on the number of parameters required.

NN2ClassifierValidation.java

```
public void train(ClassifiedBitmap[] maps, int nPresentations, double eta)
{
      int q = maps.length/2;
      int r = maps.length%2;
      int count=0;
      ClassifiedBitmap[] trainingSets = new ClassifiedBitmap[q+r];
      for (int i=0; i<trainingSets.length; i++)
            trainingSets[i] = maps[count++];
      ClassifiedBitmap[] validationSets = new ClassifiedBitmap[q];
      for (int i=0; i<validationSets.length; i++)
            validationSets[i] = maps[count++];
      double trainingError = 0.0;
      double validationError = 0.0;
      for (int p=0; p<nPresentations; p++) {
            int sample=rand.nextInt(trainingSets.length);
```

```
                trainingError += nn.train(((Bitmap)
                      trainingSets[sample]).toDoubleArray(),
                      targets[trainingSets[sample].getTarget()], eta);
                int vrand=rand.nextInt(validationSets.length);
                int actual=index((Bitmap)validationSets[vrand]);
                int target=validationSets[vrand].getTarget();
                validationError += Math.sqrt(Math.abs(actual-target)/26.0);
                System.out.println("Number of iteration: "+p);
                if (validationError == trainingError)
                      break;
          }
}
```

Performance of Extension on Training and Test Data

Algorithm	Description	Learning rate	Number of hidden nodes	Number of presentations	Training error	Test error
NN/BP	Two-layer	0.1	30	50000	397/2803	897/2803
NN/BP	Two-layer	0.1	30	50000	532/2803	666/2803
NN/BP	Two-layer	0.1	30	50000	320/2803	883/2803
NN/BP	Two-layer	0.1	30	50000	290/2803	832/2803

The implementation did not seem to have improvement on the performance. The reason is the algorithm implemented did not stop at the right time and thus the problem of overtraining was not actually prevented.

Conclusion of the Study

A better validation method should be implemented to examine the effectiveness of early stopping for preventing overtraining.

Reference

Mohan, Ravi (2010). *aima.core.learning.learners.AdaBoostLearner.java* (Online). aima-java. Java implementation of algorithms from Norvig And Russell's "Artificial Intelligence - A Modern Approach 3rd Edition.". URL: http://aima-java.googlecode.com/svn/trunk/aima-core/src/main/java/aima/core/learning/learners/AdaBoostLearner.java [Accessed 18 October 2010]

Pal, Mahesh (2007). *Ensemble Learning with Decision Tree for Remote Sensing Classification* (Online). World Academy of Science, Engineering and Technology. URL: http://www.waset.org/journals/waset/v36/v36-47.pdf [Accessed 17 October 2010]

Russel, Stuart & Norvig, Peter (2009). *Artificial Intelligence: A Modern Approach, Third Edition*, Prentice Hall.