Dana Spiegel
spiegel@media.mit.edu

6.038
5/14/97

# Gesture Recognition System

## Project Definition

The goal of this project was to build a working Gesture Recognition System. The system must be able to look at a user, and determine where they are pointing. Such a system would be integral to an "intelligent room," in which a user interacts with a computer by making verbal statements accompanied by gestures to inform the computer of, for example, the location of a newly filed, physical document.

## Design

The GRS is written in C++, with the c4.5 (which is considered an external part of this project) component written in C (meaning that with a little work, it can be linked into the main part of the system. To read in the images, and access their data, an image class (written at the AI Lab, and found on a WWW site as reusable code) based on the SGI Image Library was utilized.

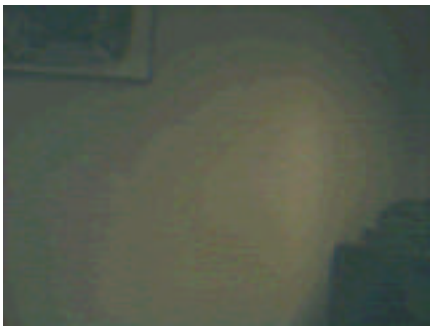

*figure 1 - a background image*

The operation of the system proceeds in four basic steps: Image input, background subtraction, image processing and data extraction, and decision tree generation/parsing (initial training of the system requires the generation of a decision tree, however subsequent use of the system only requires the parsing of the decision tree to classify the image).

To input image data into the system, an SGI IndyCam was used, with an SGI image capture program used to take the picture. The camera, which was used at an Athena workstation, was used to take first a background image (*figure 1*), and then to take subsequent images of a person pointing in particular directions. A basic assumption of the system is that these images are fairly standard: a the image is assumed to be of a person's upper body, facing forward (*figure 2*), with only one arm outstretched to a particular side.

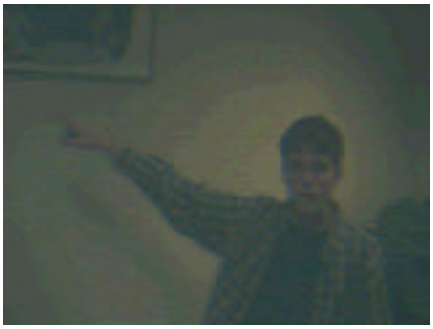

*figure 2 - a foreground image*

Once images are taken, the system performs a background subtraction of the image to isolate the person and create a mask. The background subtraction proceeds in two steps. First, each pixel

from the background image is channelwise subtracted from the corresponding pixel from the foreground image. The resulting channel differences are summed, and if they are above a threshold (in this case 20), then the corresponding image of the mask is set white, otherwise it is set black.

```
for (int j=0; j<ysize; j++)
   for (int i=0; i<xsize; i++) {

     // create mask based on cumulative differences of channels
     unsigned int temp = (Foreground->Red(i, j) - Background->Red(i, j)) +
                         (Foreground->Green(i, j) - Background->Green(i, j)) +
                         (Foreground->Blue(i, j) - Background->Blue(i, j));

     if (abs(temp) < threshold)
      temp = 0;
     else temp = 255;

     unsigned int Rout;
     unsigned int Gout;
     unsigned int Bout;

     // apply mask to the Foreground
     if (temp == 0)
      Rout = 0;
     else
      Rout = 255;
     if (temp == 0)
      Gout = 0;
     else
      Gout = 255;
     if (temp == 0)
      Bout = 0;
     else
      Bout = 255;

     // Set pixel in output image:
     Out->Pixel(i, j, PackPixel(Rout, Gout, Bout));
```

The resulting image is a mask that outlines the body of the person (*figure 3*). Two important features of the image are the existence of a second right "arm", which is the result of a shadow of the right arm falling on the wall behind the person, and the noise in the generated mask image. This phantom arm is the result of the poor image quality of the input image, but could be corrected for by the conversion of the color space of the images and the use of another method of background subtraction. If the images were converted from RGB to HSB color space, than the subtracted values of the pixels (before being set to white or black, could be inspected, and those pixels that have a very low brightness could be discarded as well (set to black). Since a shadow tends to be very dark when compared to the body of a person (in an image), those pixels that have a low brightness can be inferred to be part of a shadow, and therefore unimportant (discardable).

*figure 3 - a background subtracted mask*

The noise in the mask image can be reduced significantly by running an averaging filter over the mask data. For the GRS, I wrote a function that compares a pixel to the surrounding pixels (of a given radius), and sets that pixel to the value (black or white) of the majority of the other pixels. The GRS runs two such filters over the mask data, one with a radius of one pixel, and another of a radius of three pixels.

```
// run an averaging filter with a radius of size
for (x=1; x<xSize; x++)
  for (y=1; y<ySize; y++)
    {
     for (xtemp=x-size; xtemp<=x+size; xtemp++)
       for (ytemp=y-size; ytemp<=y+size; ytemp++)
         if (newMask->Red(xtemp, ytemp) != 0)
           avg++;

     // if most of the pixels around the current pixel are white,
     // set that pixel to white
     if (avg > (size / 2))
       mask->Pixel(x, y, PackPixel(255, 255, 255));
     else
       mask->Pixel(x, y, PackPixel(0, 0, 0));

     avg = 0;
    }
```

The final mask image is of good quality. There is no noise, and disconnected areas of the body (in this case a hand that is separated from its arm in *figure 3*) are reconnected (*figure 4*).

*figure 4 - an equalized background subtracted image*

Once a mask is generated, then that image can be processed for data to extract into a decision tree. Two strategies for extracting data from the image were tried. The first was to find the body and arms. Each column of pixels from the mask was summed, and the column with the highest sum was assumed to be the center of the body. This is a valid criteria for determining the body center, based on the assumptions of the input image. Arms are significantly thinner than the height of the main body of a person, and are so within the mask (even with the existence of shadow artifacts). The center of a person's head is the highest point on the body, and the torso of the body extends to the bottom of the image. In all of the samples this technique was tried on (66), it was successful in finding the center of the body, within a few pixels.

```
// find the vertical line that has the highest concentration of body
for (x=0; x<mask->x(); x++)
  {
    for (y=0; y<mask->y(); y++)
     if (mask->Red(x, y) != 0)
       total++;

    if (total > xMaxValue)
     {
       xMaxCoord = x;
```

```
        xMaxValue = total;
      }

    total = 0;
  }
```

From the center of the body, horizontal rows are traced out to the left and right until the edge of the mask is reached (pixels turn from white to black). The row of pixels that extends the furthest is assumed to be the arm that is pointing. This again is a valid decision based on the assumptions about the input image. Only one arm is pointing at a time, and the arm is pointing in a direction (not up or down, in which case this algorithm obviously fails). This algorithm was successful at finding the pointing arm to within a few pixel for all images tested, including those with shadows (due to the fact that the shadows are in the same direction as the pointing arm).

```
// find the maximum distances left and right from the middle, and call them arms
for (y = 0; y < mask->y(); y++)
  {
    if (mask->Red(xMiddle, y) != 0)
       {

         // left first
         for (x = xMiddle; mask->Red(x, y) != 0; x—);
         if (x < xLeftMax)
           {
            xLeftMax = x + 1;
            yLeftMax = y;
           }

         // then right
         for (x = xMiddle; mask->Red(x, y) != 0; x++);
         if (x > xRightMax)
           {
            xRightMax = x - 1;
            yRightMax = y;
           }

       }
  }
```

This algorithm works well for extracting relevant information from the image (*figure 5*). The data generated from this algorithm is output to a file as a series of values that make up a description vector for the image. Each value is reported in percent of the image, so that the system can be resolution independent (the system can be trained using images of one resolution, but can be executed using images of a different resolution).

The second algorithm that was used to extract data from the mask images was one that seems less intelligent. The mask is split into twelve columns, and the average top and bottom of the white pixels of the column is taken to be the height of that column.

```
for (x = 0; x < (mask->x() - increment); x += increment)
  {
    yTop = 0;
    yBottom = 0;
```

```
// for strip of the image, find the average size of the mask
for (xSmall = 0; xSmall < increment; xSmall++)
{
  for (y = 0; y < mask->y(); y++)
    if (mask->Red(x, y) != 0)
      yMaxBottom = y;

  for (y = mask->y(); y > 0; y—)
    if (mask->Red(x, y) != 0)
      yMaxTop = y;

  yTop += yMaxTop;
  yBottom += yMaxBottom;
}
yTop = yTop / increment;
yBottom = yBottom / increment;
```

This algorithm is good at simplifying the data presented in the mask image, and seems a somewhat intuitive description of the image,in terms of showing the body and distribution of the arms (*figure 6*). This column height is then written to file as a vector description for the image. The data is written as percentages of the height of the image, so again the algorithm is resolution independent.

*figure 5 - image illustrating body center and arm algorithim*

Once the data vectors are written to disk,c4.5 is executed to create the decision tree. This process is straightforward, involving only the manual classification of the test vectors into six categories representing the images: right-up, right-middle, right-down, left-up, left-middle, left-down. The decision to use only these six vectors was a result of the use of a decision tree, as well as the difficulty in classifying the images with more granularity. Since the system uses a decision tree, the method of learning must be supervised. Besides being tedious, the manual classification of the images is difficult to classify into more categories.

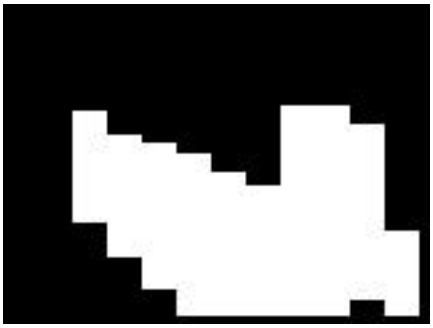

*figure 6 - a mask converted to vertical bar representation*

The system as presented here is a combination of the above two algorithms for image data extraction. The output from c4.5 gives the following decision tree:

```
bar 3 <= 0.35 :
|   left arm y <= 0.458333 :
|   |   bar 7 > 0.558333 : left-up (7.0)
|   |   bar 7 <= 0.558333 :
|   |   |   bar 3 <= 0.133333 : left-middle (7.0)
|   |   |   bar 3 > 0.133333 : left-up (5.0/1.0)
|   left arm y > 0.458333 :
|   |   bar 2 <= 0 : left-down (9.0)
|   |   bar 2 > 0 : left-middle (2.0)
```

```
bar 3 > 0.35 :
|   bar 2 <= 0 : left-up (2.0)
|   bar 2 > 0 :
|   |   bar 10 > 0.575 : right-up (11.0)
|   |   bar 10 <= 0.575 :
|   |   |   bar 5 > 0.75 : right-middle (3.0/1.0)
|   |   |   bar 5 <= 0.75 :
|   |   |   |   bar 6 <= 0.541667 : right-down (11.0/1.0)
|   |   |   |   bar 6 > 0.541667 : right-middle (9.0/1.0)
```

To use the system, instead of generating the tree,a single image is processed, and then run through the decision tree by c4.5, resulting in the classification of the image into one of the six afore mentioned categories.

## Results and Extensions

The results from the system are decent. The error from the final set of algorithms is low, 6.1%, with an expected error of 24.7%. The resulting tree, however, seems less intuitive than previous versions of the system in which only the first data extraction method (body center and arms) was used. In that case, the top of the tree was split by the location of the center of the body: if the body was on the left, then the image was that of someone pointing right. However, the final system has a lower actual error and estimated error. (There was unfortunately no time to run an error analysis to determine what kinds of errors the final system was making.

The system has many key elements. It is resolution independent,meaning that the training and the test data (or even different training data) need not be of the same resolution. The background subtraction that is used by the system is good: it effectively isolates the body of a person from the background, and does so with no visible image noise, making further processing of the image more accurate. The system is fast (the image processing takes a minimal amount of time, and c4.5 is very fast at classifying images). Since the system is based on an object class that is itself based on the SGI Image library, modifying the system to take advantage of real time images (directly from an IndyCam, for example) involves merely changing the image object class to use the SGI Movie library. The system is also fairly accurate, give decent quality images (the test images were at the bottom end of the acceptable image quality). Finally, the system makes use of both straight image processing and artificial intelligence techniques, striking a good balance between low-level understandability of the system and machine learning.

The drawbacks of the system are few but significant. First, the assumption that there is only one user and their orientation is towards the camera limit severely the practical application of the system in a real world environment. Second, the granularity of the classification of images is too low: for a computer to use gesture as an information source,the exact direction and location of the gesture (or particularly the hand with respect to the body); merely deciding that the user is pointing right and up is of limited use to a computer application.

These problems are not insurmountable, however. In the first case, a user tracking module can be used to isolate the user, as well as determine the rotation of the user with respect to the camera taking pictures of the user. This data can be used to create a more complex decision tree that can

decide the location of a user's gesture accurately in three dimensions, as well as allowing the user to roam a room. In the second case, a more AI centric approach may generate more useful data. Instead of performing a large amount of data processing on the image before using a decision tree, doing a minimal amount of data processing and then passing the image data to a neural net might produce more specific information about the location of a user's hand.

The system does work as intended, as is, therefore, a success.

## gesture.cpp

```
#include “rgbpic.h”
#include <stdio.h>
#include <fstream.h>


int
main(int argc, char *argv[])
{
  int y;
  line body, arm;


  if (argc != 3) {
    cerr << “This program subtracts the background of an image” << endl;
    cerr << “usage: “ << argv[0] << “ background.rgb foreground.rgb” << endl;
    exit(1);
  }

  // Load .rgb file into new RGBPic object:
  RGBPic *Background = new RGBPic(argv[1]);
  RGBPic *Foreground = new RGBPic(argv[2]);

  // Exit if unable to load:
  if (Background->x() == 0 || Foreground->x() == 0)
    return 1;

  // Exit if the sizes of the input images are not the same
  if ((Background->x() != Foreground->x()) || (Background->y() != Foreground->y()))
    {
      cerr << “The sizes of the images aren’t the same!” << endl;
      return 1 ;
    }

  // Create new RGBPic:
  RGBPic *Out = new RGBPic();
  Out->New(Background->x(), Background->y());

  if (!(Subtract(Background, Foreground, Out, 20)))
    {
      cerr << “Couldn’t do background subtraction!” << endl;
      return 1;
    }
  Out->SaveRGB(“subtracted.rgb”);

  // Clean up the noise in the image
  EqualizeMask(Out, 1);
  EqualizeMask(Out, 3);

  Out->SaveRGB(“equalized.rgb”);

  // Open the data file
  ofstream datafile (“gesture.data”, ios::app);

  // Find Vertical Bars
  VerticalBars(Out, &datafile);

  // Find the Body of the person in the image and output the data
```

```
  body = FindBodyCenter(Out, &datafile);
  arm = FindMostWidth(Out, body.start.x, body.start.y, &datafile);

  // Close the data file
  datafile.close();

  // Draw in the center line (for visual purposes only
  for (y = body.start.y; y < body.end.y; y++)
    Out->Pixel(body.start.x, y, PackPixel(0, 255, 0));

  // Save output image:
  Out->SaveRGB("bodyandarm.rgb");

  return 0;
}


int
Subtract(RGBPic *Background, RGBPic *Foreground, RGBPic *Out, int threshold)
  // a better background subtraction routine would also cut out pixels that
  // had a low saturation, but that would involve converting from an RGB
  // color model to an HSB model (not something I want to worry about now).
{
  // Get size of picture:
  int xsize = Background->x();
  int ysize = Background->y();

  for (int j=0; j<ysize; j++)
    for (int i=0; i<xsize; i++) {

      // create mask based on cumulative differences of channels
      unsigned int temp = (Foreground->Red(i, j) - Background->Red(i, j)) +
                          (Foreground->Green(i, j) - Background->Green(i, j)) +
                          (Foreground->Blue(i, j) - Background->Blue(i, j));

      if (abs(temp) < threshold)
       temp = 0;
      else temp = 255;

      unsigned int Rout;
      unsigned int Gout;
      unsigned int Bout;

      // apply mask to the Foreground
      if (temp == 0)
       Rout = 0;
      else
       Rout = 255;
      if (temp == 0)
       Gout = 0;
      else
       Gout = 255;
      if (temp == 0)
       Bout = 0;
      else
       Bout = 255;

      // Set pixel in output image:
      Out->Pixel(i, j, PackPixel(Rout, Gout, Bout));
    }
```

```
  return 1;
}


PixelType
PackPixel(int red, int green, int blue)
{
  PixelType p = (red) | (green << 8) | (blue << 16);
  return p;
}


int
EqualizeMask(RGBPic *mask, int size)
{
  int xSize = mask->x() - size, ySize = mask->y() - size;
  int x = size, y = size;
  int xtemp, ytemp;
  unsigned int avg = 0;
  RGBPic *newMask = new RGBPic(*mask);

  if (newMask == NULL)
    return 0;
  if (mask == NULL)
    return 0;


  // run an averaging filter with a radius of size
  for (x=1; x<xSize; x++)
    for (y=1; y<ySize; y++)
      {
       for (xtemp=x-size; xtemp<=x+size; xtemp++)
         for (ytemp=y-size; ytemp<=y+size; ytemp++)
           if (newMask->Red(xtemp, ytemp) != 0)
             avg++;

       // if most of the pixels around the current pixel are white,
       // set that pixel to white
       if (avg > (size / 2))
         mask->Pixel(x, y, PackPixel(255, 255, 255));
       else
         mask->Pixel(x, y, PackPixel(0, 0, 0));

       avg = 0;
      }

  // clean up left and right sides
  for (x=0; x<size; x++)
    for (y=0; y<mask->y(); y++)
      mask->Pixel(x, y, PackPixel(0, 0, 0));
  for (x=mask->x(); x>mask->x() - size; x—)
    for (y=0; y<mask->y(); y++)
      mask->Pixel(x, y, PackPixel(0, 0, 0));

  // clean up top and bottom
  for (y=0; y<size; y++)
    for (x=0; x<mask->x(); x++)
      mask->Pixel(x, y, PackPixel(0, 0, 0));
  for (y=mask->y(); y>mask->y() - size; y—)
```

```
    for (x=0; x<mask->x(); x++)
      mask->Pixel(x, y, PackPixel(0, 0, 0));

  return 1;
}


line
FindBodyCenter(RGBPic *mask, ofstream *datafile)
{
  int total=0, x, y;
  int xMaxValue=0, xMaxCoord=0;
  line body;

  // find the vertical line that has the highest concentration of body
  for (x=0; x<mask->x(); x++)
    {
      for (y=0; y<mask->y(); y++)
       if (mask->Red(x, y) != 0)
         total++;

      if (total > xMaxValue)
       {
         xMaxCoord = x;
         xMaxValue = total;
       }

      total = 0;
    }

  // set the x coordinates to that center line
  body.start.x = xMaxCoord;
  body.end.x = xMaxCoord;

  // output the location of the center of the body (left, right, or middle)
  if (xMaxCoord < (mask->x()/2))
    *datafile << "l, ";
  else if (xMaxCoord > (mask->x()/2))
    *datafile << "r, ";
  else if (xMaxCoord = (mask->x()/2))
    *datafile << "m, ";

  // Find the top and bottom of the body
  for (y = 0; (mask->Red(body.start.x, y) == 0) && (y < mask->y()); y++);
  body.end.y = mask->y() - y;
  for (y = mask->y(); (mask->Red(body.start.x, y) == 0) && (y > 0); y—);
  body.start.y = mask->y() - y;

  // output the top of the body (in percentage of height)
  *datafile << (float)body.start.y / (float)mask->y() << ", ";

  return body;
}


line
FindMostWidth(RGBPic *mask, int xMiddle, int yTop, ofstream *datafile)
{
  int x, y;
  int xLeftMax = xMiddle, yLeftMax = 0, xRightMax = xMiddle, yRightMax = 0;
```

```
    line arm;

    // find the maximum distances left and right from the middle, and call them arms
    for (y = 0; y < mask->y(); y++)
      {
        if (mask->Red(xMiddle, y) != 0)
           {

             // left first
             for (x = xMiddle; mask->Red(x, y) != 0; x—);
             if (x < xLeftMax)
               {
                xLeftMax = x + 1;
                yLeftMax = y;
               }

             // then right
             for (x = xMiddle; mask->Red(x, y) != 0; x++);
             if (x > xRightMax)
               {
                xRightMax = x - 1;
                yRightMax = y;
               }

           }
      }

    // find the coordinates for the hands of the arm that is the longest
    // this is the arm we are interested in positioning
    if ((xRightMax - xMiddle) > (xMiddle - xLeftMax))
      {
        arm.start.x = xMiddle;
        arm.start.y = yRightMax;
        arm.end.x = xRightMax;
        arm.end.y = yRightMax;
      }
    else if ((xRightMax - xMiddle) < (xMiddle - xLeftMax))
      {
        arm.start.x = xLeftMax;
        arm.start.y = yLeftMax;
        arm.end.x = xMiddle;
        arm.end.y = yLeftMax;
      }

    // draw in the arms
    for (x = xLeftMax; x <= xMiddle; x++)
      mask->Pixel(x, yLeftMax, PackPixel(0, 255, 0));
    for (x = xRightMax; x >= xMiddle; x—)
      mask->Pixel(x, yRightMax, PackPixel(0, 255, 0));

    // output the data from the mask (in percentage of height and width
    // left arm
    *datafile << (float)(xMiddle - xLeftMax) / (float)mask->x() << “, “;
    *datafile << (float)((120 - yLeftMax) - yTop) / (float)mask->y() << “, “;
    // right arm
    *datafile << (float)(xRightMax - xMiddle) / (float)mask->x() << “, “;
    *datafile << (float)((120 - yRightMax) - yTop) / (float)mask->y() << endl;

    return arm;
  }
```

```
void
VerticalBars(RGBPic *mask, ofstream *datafile)
{
  int x, y, xSmall, yMaxTop = 0, yMaxBottom = 0;
  int yTop, yBottom;
  RGBPic *temp = new RGBPic(*mask);
  int increment = mask->x() / 12; // each strip = 1/12 of the width of the image
                                  // (no particular reason, but it works!)

  for (x = 0; x < (mask->x() - increment); x += increment)
    {
      yTop = 0;
      yBottom = 0;

      // for strip of the image, find the average size of the mask
      for (xSmall = 0; xSmall < increment; xSmall++)
       {
         for (y = 0; y < mask->y(); y++)
           if (mask->Red(x, y) != 0)
             yMaxBottom = y;

         for (y = mask->y(); y > 0; y—)
           if (mask->Red(x, y) != 0)
             yMaxTop = y;

         yTop += yMaxTop;
         yBottom += yMaxBottom;
       }
      yTop = yTop / increment;
      yBottom = yBottom / increment;

      // output width data for the bar (in percent of vertical size)
      *datafile << (float)(yBottom - yTop) / (float)mask->y() << “, “;

      // make this strip of the temp image completely black
      for (xSmall = 0; xSmall < increment; xSmall++)
       for (y = 0; y < mask->y(); y++)
         temp->Pixel(x + xSmall, y, PackPixel(0, 0, 0));

      // draw in the bar for this strip
      for (xSmall = 0; xSmall < increment; xSmall++)
       for (y = yTop; y < yBottom; y++)
         temp->Pixel(x + xSmall, y, PackPixel(255, 255, 255));
    }

  temp->SaveRGB(“VerticalBars.rgb”);
}
```

## gesture.names

```
left-up, left-middle, left-down, right-up, right-middle, right-down. | classes
bar 1:              continuous.
bar 2:              continuous.
bar 3:              continuous.
bar 4:              continuous.
bar 5:              continuous.
bar 6:              continuous.
```

```
bar 7:              continuous.
bar 8:              continuous.
bar 9:              continuous.
bar 10:             continuous.
bar 11:             continuous.
bar 12:             continuous.
body x :            l, r, m.
body start y :      continuous.
left arm x :        continuous.
left arm y :        continuous.
right arm x :       continuous.
right arm y :       continuous.
```

## Output from c4.5 training

```
C4.5 [release 6] decision tree generator Mon May 12 01:05:02 1997
-----------------

    Options:
      File stem <gesture>

Read 66 cases (18 attributes) from gesture.data

Decision Tree:

bar 3 <= 0.35 :
|   left arm y <= 0.458333 :
|   |   bar 7 > 0.558333 : left-up (7.0)
|   |   bar 7 <= 0.558333 :
|   |   |   bar 3 <= 0.133333 : left-middle (7.0)
|   |   |   bar 3 > 0.133333 : left-up (5.0/1.0)
|   left arm y > 0.458333 :
|   |   bar 2 <= 0 : left-down (9.0)
|   |   bar 2 > 0 : left-middle (2.0)
bar 3 > 0.35 :
|   bar 2 <= 0 : left-up (2.0)
|   bar 2 > 0 :
|   |   bar 10 > 0.575 : right-up (11.0)
|   |   bar 10 <= 0.575 :
|   |   |   bar 5 > 0.75 : right-middle (3.0/1.0)
|   |   |   bar 5 <= 0.75 :
|   |   |   |   bar 6 <= 0.541667 : right-down (11.0/1.0)
|   |   |   |   bar 6 > 0.541667 : right-middle (9.0/1.0)


Tree saved


Evaluation on training data (66 items):

       Before Pruning           After Pruning
      ----------------   ---------------------
      Size      Errors   Size      Errors   Estimate

        19    4( 6.1%)     19    4( 6.1%)    (24.7%)   <<
```