MATH 444 Final Project

Connor Wolfe

May 2017

Introduction

This assignment develops skills in building classification trees and then optimally pruning the tree to yield the most accurate classification. We will first introduce the ideas and algorithms behind classification trees and how to prune them. Next we will demonstrate findings on classifying the 'Irisdata.mat' set and the 'GlassData.mat' set.

Data Sets

The 'GlassData.mat' contains the data set X (9x214) and annotation vector I (1x214). The 9-dimensional data is classified by its refractive index, and weight percent of Na, Mg, Al, Si, K, Ca, Ba, Fe. The annotation vector stores the class of each data point $I \epsilon [1, 2, 3, 4, 5, 6]$.

Data Storage

We will use structs to represent the trees. The structs are stored in array R and sorted by index. They will have the following properties:

```
R(j).I= list of data point indices included in the rectangle
R(j).p= Frequency of each class
R(j).j= The dimension of the data that is optimal to split
R(j).s=  Value to split the rectangle into two optimally classified children
R(j).left= Index of left child
R(j).right= Index of right child
```

It is clear that when a rectangle has no children, we assign R(j).left=NaN and R(j).right=NaN, R(j).s=NaN, and R(j).j=NaN.

Helper Functions

ClassDistr

Function to determine the frequency of each class of a given rectangle, the percent it misclassifies, and the majority class

Input: C=vector of classes of data
I= index vector corresponding to indices of data in rectangle in I
Output: p= vector of frequencies of each class (note the sum of elements in p=1)
c= majority class (all data in this rectangle will be classified as c)
r=misclassification error (rate that we misclassify points in this rectangle)

Implementation:

```
function [p, c, r]=ClassDistr(C,I)
    n=size(I,2);  %total number of elements in the rectangle
    c=mode(C(I)); %c is the mode of the classes of the elements in the rectangle
    classes=unique(C); %find the class values
    k=size(classes,2); %and the number of classes to iterate through
    p=zeros(1,k);  %the size of p must be the number of classes
    for i=1:k
        num_i=size(find(C(I)==classes(i)),2); %find number of elements in class i
        p(i)=num_i/n; %divide this by the total number of elements for the frequency
    end
    r=1-p(c); %the misclassification error is 1 minus the frequency of the majority class,
    %since all other elements are misclassified as this majority class
end
```

OptimalSplit

Function to return the data dimension and value to split the rectangle at in order to optimally classify the data
Input: I_ind=index vector corresponding to indices of data in rectangle in I
C=vector of classes of data
X=Data matrix
Output: j=dimension of x to split on
s=value of x to split above and below

Implementation:

```
We must iterate through all dimensions of X and find within each dimension, which value
s value would yield the best split.  We will store split values for each dimension in
vector m\_j which stores the optimal split value and the mismatch it yields for each
dimension. In order to calculate the split value given a dimension j, we do the following:
(a) Find the unique values of the data and sort them in increasing order.  We can now
observe that the only splits which make sense are those between the sorted X points.
In matlab we write:
    [x_sort, j_sort] = sort(x, 'ascend');
(b) Store every reasonable split value in vector s, where it will store the median
between X(i) and X(i+1):
        s(i)=0.5 * (x_sort(i) + x_sort(i+1));
(c) Sort the data points about the split value to yield the indices of the points
left and right of it, and the class values of these points.
```

```
        I_left=I(find(X(j,I) <= s(i)));
        c_left=C(I_left);
The same implementation holds for the right side.

(d) Calculate the mean of the values to the left and to the right
        c1=(1/(length(c_left)))*sum(c_left);
(e) Calculate the mismatch of all X as its variance from the mean of the rectangle
we sort it in:
        m_s(i)=sum((c_left - c1).^2) + sum((c_right-c2).^2);
(f) Return the split that minimizes the mismatch for this dimension and add it
to the m_j vector
    [m_val, ind]=min(m_s);
    m_j(:,j)=[m_val;s_star];
(g) Once we have calculated the split for each j, we again minimize the mismatch
within j to find the best j and s values
    [~, j_opt] = min(m_j(1,:));
    s_opt=m_j(2, j_opt);
(h) return j_opt and s_opt
```

MisclassCost

Input: R=rectangle to analyze
n= total number of data points in the set
C= classification vector
Output: m= misclassification cost of the rectangle weighted by its size
Implementation:
First, we calculate the r value for rectangle R with the ClassDistr function, named r_r
Next, we calculate the weight of the node as the number of elements inside it divided by the total number of elements:
v_r=length(R.I)/n;
Last, m is simply the product of v_r and r_r
m=v_r*r_r;

Building Classification Trees

Idea

A classification tree, as represented in figure (1) is a set of divisions of data into increasingly small rectangles. The rectangle edges are calculated using the OptimalSplit() function to be the points which will best separate the data into its constituent classes. When a rectangle is divided, the data inside is separated into the two child rectangles, left and right, depending on if the data is less

than or greater than the threshold point, respectively. Each child rectangle is assigned a class value based on the majority vote of the elements which compose it, and all data points inside will be approximated to this majority vote class. It is important than that the rectangles capture the data well so that when we classify using their majority, we do not misclassify too many points.
I will now describe the procedure for generating a classification tree that terminates with all pure leaves. This procedure will begin with the entire data set R(1), and will continue to divide it using OptimalSplit() until all leaves are pure. This tree rooted at R(1) and ending with all pure leaves we will call T_{max}

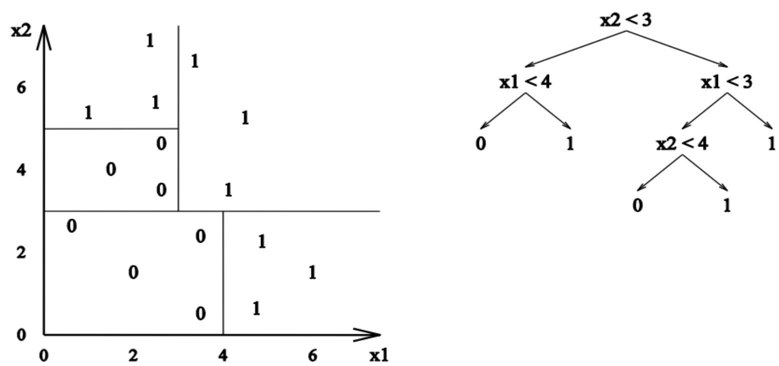


Figure 1: An example of a classification tree for unrepresentative data. We see the original tree T_0 divided into two smaller rectangles, points such that $x_2 <= 3$ or points such that $x_2 > 3$. The graph continues to divide until all of the rectangles are of the same class (pure), yielding T_{max}.

Algorithm

1. Initialize: Let R(1) be the rectangle containing all the data. We set R(1).I=I
Let PureNodes=[ ] be an empty vector that will store the pure nodes
Let MixedNodes= [1] be a vector containing indices of the mixed nodes
count=1;
2. While length(MixedNodes)>0
(a) Pick the next node, split it, and update its fields

```
ind=MixedNodes(1);
I_ind=R(ind).I;
X_ind=X(:,I_ind);
[j,s]=OptimalSplit(I_ind,C,X);
R(ind).j=j;
R(ind).s=s;
R(ind).left=count+1;
R(ind).right=count+2;
```

(b) Define the children based on the split values of the parent (shown for left only)

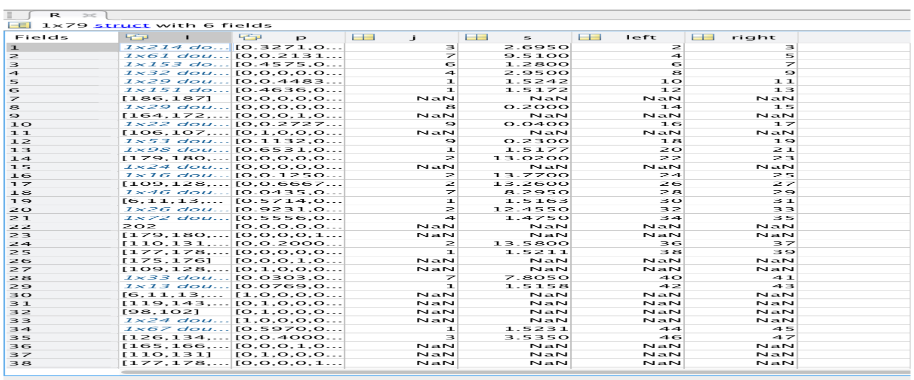
```
I_left=I_ind(find(X_ind(j,:)<=s));  %The index values for the child are those correspond
[p_left, c_left, r_left]=ClassDistr(C,I_left);  %With the index values we can find the p
R(count+1).I=I_left;  %update its fields
R(count+1).p=p_left;
R(count+1).j=NaN;
R(count+1).s=NaN;
R(count+1).left=NaN;
R(count+1).right=NaN;
```

(c) Check the purity of the children and update the pure/mixed arrays

```
if r_left==0  %if the misclassification is 0, it is pure
    PureNodes=[PureNodes, count+1];
else
    MixedNodes=[MixedNodes, count+1];
end
MixedNodes=MixedNodes(2:end) %remove the node we just split
count=count+2;
```

Results

Figure(2) shows a data table representing the array R of nodes. In building the tree, we created the maximally pruned tree, therefore all leaves are pure.


R

1x79 struct with 6 fields

Fields	I	p	j	s	left	right
1	1x214 do...	[0.3271,0...	3	2.6950	2	3
2	1x61 dou...	[0,0.2131...	7	9.5100	4	5
3	1x153 do...	[0.4575,0...	6	1.2800	6	7
4	1x32 dou...	[0,0,0,0,0.0...	4	2.9500	8	9
5	1x29 dou...	[0,0.4483...	1	1.5242	10	11
6	1x151 do...	[0.4636,0...	1	1.5172	12	13
7	[186,187]	[0,0,0,0,0,0...	NaN	NaN	NaN	NaN
8	1x29 dou...	[0,0,0,0,0,0...	8	0.2000	14	15
9	[164,172,...	[0,0,0,1,0...	NaN	NaN	NaN	NaN
10	1x22 dou...	[0,0.2727...	9	0.0400	16	17
11	[106,107,...	[0,1,0,0,0...	NaN	NaN	NaN	NaN
12	1x53 dou...	[0.1132,0...	9	0.2300	18	19
13	1x98 dou...	[0.6531,0...	1	1.5177	20	21
14	[179,180,...	[0,0,0,0,0,0...	2	13.0200	22	23
15	1x24 dou...	[0,0,0,0,0,0...	NaN	NaN	NaN	NaN
16	1x16 dou...	[0,0.1250...	2	13.7700	24	25
17	[109,128,...	[0,0.6667...	2	13.2600	26	27
18	1x46 dou...	[0.0435,0...	7	8.2950	28	29
19	[6,11,13,...	[0.5714,0...	1	1.5163	30	31
20	1x26 dou...	[0.9231,0...	2	12.4550	32	33
21	1x72 dou...	[0.5556,0...	4	1.4750	34	35
22	202	[0,0,0,0,0,0...	NaN	NaN	NaN	NaN
23	[179,180,...	[0,0,0,0,1...	NaN	NaN	NaN	NaN
24	[110,131,...	[0,0.2000...	2	13.5800	36	37
25	[177,178,...	[0,0,0,0,0,0...	1	1.5211	38	39
26	[175,176]	[0,0,0,1,0...	NaN	NaN	NaN	NaN
27	[109,128,...	[0,1,0,0,0...	NaN	NaN	NaN	NaN
28	1x33 dou...	[0.0303,0...	7	7.8050	40	41
29	1x13 dou...	[0.0769,0...	1	1.5158	42	43
30	[6,11,13,...	[1,0,0,0,0,0...	NaN	NaN	NaN	NaN
31	[119,143,...	[0,1,0,0,0...	NaN	NaN	NaN	NaN
32	[98,102]	[0,1,0,0,0...	NaN	NaN	NaN	NaN
33	1x24 dou...	[1,0,0,0,0,0...	NaN	NaN	NaN	NaN
34	1x67 dou...	[0.5970,0...	1	1.5231	44	45
35	[126,134,...	[0,0.4000...	3	3.5350	46	47
36	[165,166,...	[0,0,0,1,0...	NaN	NaN	NaN	NaN
37	[110,131]	[0,1,0,0,0...	NaN	NaN	NaN	NaN
38	[177,178,...	[0,0,0,0,1...	NaN	NaN	NaN	NaN

Figure 2: The Array R of structs representing all 79 nodes and leaves that define T_max. To prove this is in fact T_max, the sum of all the indices of pure nodes is 214, therefore all data points are captured in a pure node. We see for instance, that the original node containing all data points can be optimally split in its 3rd dimension by a value 2.695, which will yield its left child at index 2 and right child at index 3.

Pruning the Tree

Idea

While for the training set T_max will perform optimally and properly classify all data points, this tree will often be overfitted and will not capture the more general differences between data points for when new points are to be tested. Pruning a tree is when we take a node that is not a leaf, delete its children, and make it a leaf. In our case all leaves were pure and all non-leaf nodes were mixed. Therefore, in pruning the tree we will no longer be classifying data only with pure rectangles, but instead will look for the optimal nodes, pure and mixed, that classify the data best. We need a measure to determine the effectiveness of a node of classifying, which we define as the cost-complexity measure of the tree.
The cost-complexity measure seeks a balance between the misclassification error and the complexity of a tree. We define the cost complexity measure as follows:

$$\rho_\alpha(T) = \rho(T) + \alpha|T|$$

where $\rho(T)$ is the misclassification error and $\alpha|T|$ is the complexity. We define

$$\rho(T) = \sum_{R\epsilon[leaf]} v(R)r(R)$$

such that v(r) is the weight of the leaf relative to the tree:

$$v(r) = n(R)/n(R(1))$$

and r(R) is the misclassifcation defined previously.
Lastly, the complexity |T| is defined as the number of leaves in the tree

We can calculate the difference between the cost complexity of an original tree T and the tree T-T' (where T' is rooted at $\bar{R}$) as follows:

$$\rho_\alpha(T - T') - \rho_\alpha(T) = v(\bar{R})r(\bar{R}) + \alpha - \sum_{leavesT'} v(R)r(R) + \alpha|T' \tag{1}$$

An optimal tree is one which will minimize the cost-complexity function. Note the value of α determines the weighting of the complexity versus the error. A small α will favor complex trees as the complexity has low cost, but as α increases, tress will need to be simpler to minimize $\rho_\alpha(T)$

Our algorithm will begin with T_max and will iterate through all nodes of T_max and calculate their $\rho_\alpha(T)$ to determine the best node to prune at. This yields a new pruned tree T' and its corresponding α value. We continue to prune until we arrive at the tree with only the root node included.

Algorithm

1. Initialize: Create a cell T to store all the pruned trees, and set T(1)=T_max. Set counter k=0 2. Iterate: Until $T_k = R_0$

(a) Compute the genealogy matrix $G^{m,m}$ such that if $G_{i,j} = 1$ node j is a descendant of node i.
(b) For all $\bar{R}$ nodes that are not leaves, calculate the critical point $\alpha(\bar{R})$ where pruning tree at $\bar{R}$ has a cost less than the original tree. We calculate this using eq (1) above.

$$\alpha(\bar{R}) = \frac{v(\bar{R})r(\bar{R}) - \sum_{leavesT'} v(R)r(R)}{|T'| - 1}$$

We use the MisclassCost() function to find the v(R)r(R) values. (c) Prune the tree at the node $\bar{R}_{prune}$ corresponding to the minimum α.
To do so, first we find all nodes that are descendants of $\bar{R}_{prune}$ using G

```
children=find(G(ind,:)==1);
```

Next, starting from the last child down to the first (in order to preserve ordering), we remove each child from the tree

```
R_curr=R_curr([ 1:children(c)-1, children(c)+1:length(R_curr) ]);
```

Last, we update the children pointers in the remaining nodes because the indices of nodes have changed following the deletions of leaves. To do so, we decrement every node's pointer to a child by 1 for every time the pointer is greater than the index of one of $\bar{R}_{prune}$'s children

(d) Increment k and proceed

Results

We now have a collect of pruned trees stored in T. Which pruning is ideal depends on the test set, so we will calculate the specificity for all pruned trees and find the best. I will examine the results for the 'IrisData.mat' set and the 'GlassData.mat' set.

Iris Data

I trained the data with the first 25 points of each class and tested with the remaining 25 of each class (75 points per test). Using this set, I grew the tree to T_max as described above. The tree T_max can be seen in figure (3) specificity × T × R ×

1x11 struct with 6 fields

Fields	l	p	j	s	left	right
1	1x75 dou...	[0.3333,0...	3	2.6000	2	3
2	1x25 dou...	[1,0,0]	NaN	NaN	NaN	NaN
3	1x50 dou...	[0,0.5000...	4	1.6500	4	5
4	1x25 dou...	[0,0.0400...	3	4.9500	6	7
5	1x25 dou...	[0,0.9600...	3	4.8500	8	9
6	1x24 dou...	[0,0,1]	NaN	NaN	NaN	NaN
7	70	[0,1,0]	NaN	NaN	NaN	NaN
8	[57,121]	[0,0.5000...	1	5.4000	10	11
9	1x23 dou...	[0,1,0]	NaN	NaN	NaN	NaN
10	57	[0,1,0]	NaN	NaN	NaN	NaN
11	121	[0,0,1]	NaN	NaN	NaN	NaN
12						
13						

Figure 3: The Array R of structs representing all 11 nodes (rectangles) of T_max. We see that we quickly distinguished all of class one in the first split, yielding the pure node 2, and mixed node 3, and the remaining splits tried to separate class 2 from 3. We can separate 96% pure classes in one more split, and spend the remaining 2 splits separating the one fringe point.

Next, I calculated the optimal pruning points to yield 5 increasingly pruned trees, shown in figure (4).

VARIABLE SELECTION

specificity × T ×

1x11 cell

	1	2	3	4	5
1	1x9 struct	1x7 struct	1x5 struct	1x3 struct	1x1 struct
2					
3					
4					
5					
6					
7					
8					
9					
10					
11					
12					
13					
14					
15					
16					

Figure 4: The Matlab references to the increasingly more pruned trees that began at T_max. T_max pruned two children to obtain T1, which continued to prune until we arrive at the tree with just the root node. The optimal tree to classify with depends on the data as we explore below.

The incoming data set determines which tree is best to classify with, so you must test on all trees. Below are the specificity values for the remaining iris data points classified by each tree.

specificity

1x11 double

	1	2	3	4	5
1	0.9200	0.9467	0.9467	0.6667	0.3333
2					
3					

Figure 5: The specificity values for the iris data testset classified with each tree T[1:5]. Pruning at trees 1,2, or 3 proves highly effective with specificity values above 90%. Trees 4 and 5 represent highly pruned trees with T5 only being the root node, so it is clear that these would not be effective classifiers.

Glass Data

I trained the Glass Data with I_{train}=[1:35, 71:108, 147:155, 164:170, 177:181, 186:200] and tested with I_{test}=[36:70, 109:146, 156:163, 170:176, 181:185, 200:214].
First we observe T_{max} in figure (6).

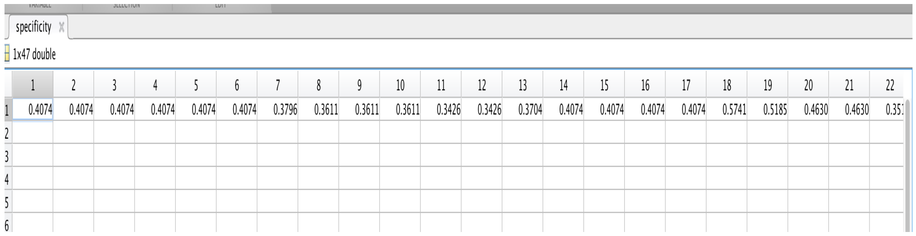
R | specificity

1x47 struct with 6 fields

Fields	I	p	j	s	left	right
1	1x109 do...	[0.3211,0...	3	2.7000	2	3
2	1x27 dou...	[0,0.1111...	2	13.5300	4	5
3	1x82 dou...	[0.4268,0...	6	1.2800	6	7
4	[106,107,...	[0,0.3333...	1	1.5232	8	9
5	1x18 dou...	[0,0,0,0,0.0...	3	2.2200	10	11
6	1x80 dou...	[0.4375,0...	1	1.5226	12	13
7	[186,187]	[0,0,0,0,0,0...	NaN	NaN	NaN	NaN
8	[165,166,...	[0,0,0,1,0...	NaN	NaN	NaN	NaN
9	[106,107,...	[0,1,0,0,0,0...	NaN	NaN	NaN	NaN
10	1x14 dou...	[0,0,0,0,0,0...	4	1.7250	14	15
11	[164,177,...	[0,0,0,0,0.2...	1	1.5167	16	17
12	1x77 dou...	[0.4545,0...	1	1.5171	18	19
13	[104,105,...	[0,0.6667...	1	1.5236	20	21
14	[180,181,...	[0,0,0,0,0,0...	1	1.5211	22	23
15	1x11 dou...	[0,0,0,0,0,0...	NaN	NaN	NaN	NaN
16	164	[0,0,0,1,0...	NaN	NaN	NaN	NaN
17	[177,178,...	[0,0,0,0,0,1...	NaN	NaN	NaN	NaN
18	1x35 dou...	[0.1143,0...	7	8.2950	24	25
19	1x42 dou...	[0.7381,0...	7	8.8950	26	27
20	188	[0,0,0,0,0,0...	NaN	NaN	NaN	NaN
21	[104,105]	[0,1,0,0,0,0...	NaN	NaN	NaN	NaN
22	[180,181]	[0,0,0,0,0,1...	NaN	NaN	NaN	NaN
23	190	[0,0,0,0,0,0...	NaN	NaN	NaN	NaN
24	1x26 dou...	[0.1538,0...	9	0.1900	28	29
25	[93,99,10...	[0,0.3333...	5	73.0500	30	31
26	1x33 dou...	[0.8788,0...	3	3.4500	32	33
27	[18,22,91...	[0.2222,0...	2	14.3400	34	35
28	1x23 dou...	[0.0435,0...	7	7.8050	36	37
29	[6,11,13]	[1,0,0,0,0,0...	NaN	NaN	NaN	NaN
30	[148,149,...	[0,0,1,0,0...	NaN	NaN	NaN	NaN
31	[93,99,101]	[0,1,0,0,0,0...	NaN	NaN	NaN	NaN
32	[96,100]	[0,1,0,0,0,0...	NaN	NaN	NaN	NaN
33	1x31 dou...	[0.9355,0...	2	13.5100	38	39
34	[91,97,98...	[0,0.7143...	2	13.3300	40	41
35	[18,22]	[1,0,0,0,0,0...	NaN	NaN	NaN	NaN
36	[3,71,85]	[0.3333,0...	1	1.5160	42	43
37	1x20 dou...	[0,1,0,0,0,0...	NaN	NaN	NaN	NaN
38	1x25 dou...	[1,0,0,0,0,0...	NaN	NaN	NaN	NaN

Figure 6: The Array R of structs representing all 47 nodes (rectangles) of T_max for the glass data set. Note that when we trained with the entire test set, there were 79 nodes, but since the data is simpler, so is T_max.

Next, I pruned the tree, yielding 22 trees of decreasing size. I tested each tree using the I_{test} set and obtained the following specificity:


specificity

1x47 double

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
1	0.4074	0.4074	0.4074	0.4074	0.4074	0.4074	0.3796	0.3611	0.3611	0.3611	0.3426	0.3426	0.3704	0.4074	0.4074	0.4074	0.4074	0.5741	0.5185	0.4630	0.4630	0.35
2																						
3																						
4																						
5																						
6																						

Figure 7: The specificity values for the glass data testset classified with each tree T[1:22]. We see that the glass data trees performered far worse with values typically between 40% and 50%. I believe this to be a fault of the data due to the high performance of the iris set.