MA 751 Final Project- Classification Trees

Anthea Cheung

April 29, 2016

Abstract

Decision trees are popular machine learning tools that make predictions by partitioning the feature space into boxes. They are conceptually simple and easy to interpret but also very powerful. In this report, I will present and overview of classification trees. An implementation of classification trees is provided and with an analysis of a numerical example in cardiac imaging. Final, some limitations with classification trees and possible solutions will be presented.

1 Methodology

1.1 Growing a Classification Tree

Suppose we have a training sample $\mathcal{T} = \{(x_i, y_i)\}_{i=1}^N$, where $x_i \in \mathbb{R}^l$ and $y_i = 1, 2, \cdots K$ is one of K classes. To the grow the classification tree, we will apply successive splits in the feature space of the form $x_j \leq s$ and $x_j > s$, where x_j is the jth component of the feature vector x.

At each split, the left child will correspond to $x_j \leq s$ and the right child will correspond to $x_j > s$. At each step, we will add two children to a leaf, thus partitioning the feature space. Thus, any feature vector will be correspond to one of the leaves of the classification tree. We will classify all vectors in the same node m as the most likely class in that node. For node m corresponding to a region R_m containing N_m observations, we will let

$$\hat{p}_{mk} = \frac{1}{N_m} \sum_{x_i \in R_m} I(y_i = k) \tag{1}$$

Note that $\hat{p}_{mk} = P(y_i = k | x_i \in R_m)$ Then we will classify vectors in R_m to class $k(m)$, where

$$k(m) = \arg\max_k \hat{p}_{mk} \tag{2}$$

After each split, we need to decide when the next "best" split would be. One possible way of deciding this would be to find the next split for node m is one that minimizes the misclassification cost:

$$\frac{1}{N_m}\sum_{i\in R_m} I(y_i \neq k(m)) = 1 - \hat{p}_{mk(m)} \tag{3}$$

1.1.1 Impurity Measures

This misclassification cost introduced above is an example of an impurity measure used to help grow our decision tree. In general, we can introduce any appropriate impurity measure $i(m)$ of a node m. After making a split on node m, the change in impurity will be:

$$\Delta i = i(t) - p_l * i(m_l) + p_r * i(m_r) \tag{4}$$

where m_l, m_r are the left and right children of node m, and p_l, p_r are the probabilities of a case in the training set belonging in m_l or m_r.

The impurity measures are chosen such that $\Delta i \geq 0$. Thus we keep choosing the next split that gives us the greatest change in impurity. In other words, we want to minimize the impurity of the left and right children: $p_l * i(m_l) + p_r * i(m_r)$. When $i(m) = 0$, we say that node m is a pure node, which means that there are no further splits on that node that will decrease the impurity. For a tree T, we will define the impurity of the tree to be

$$i(T) = \sum_{m\in\widetilde{T}} i(m) \tag{5}$$

where $\widetilde{T}$ is the set of leaves of the tree T. As we build our classification tree, we will continue introducing splits until each terminal node is pure.

Some other examples of impurity measures include the Gini index:

$$\sum_{k'\neq k} \hat{p_{mk}}\hat{p_{mk'}} = \sum_{k=1}^{K} \hat{p}_{mk}(1-\hat{p}_{mk}) = 1 - \sum_{k=1}^{K} \hat{p}_{mk}^2 \tag{6}$$

and cross-entropy:

$$-\sum_{k=1}^{K} \hat{p}_{mk}\log\hat{p}_{mk} \tag{7}$$

To choose or splits, we simply consider all components of our feature vectors over all values in the feature space. Note that for two values $s_1 < s_2$, as long as there are no training samples with $s_1 < x_ij < s_2$, the splits $x_j \leq s_1$ and $x_j > s_2$ will produce the same result. Thus we need only consider all values of variable j in the training set $\mathcal{T}$. For ordered variables, this will give us up to N different splits. For categorical variables with q possible values, however, this can give us up to $2^{q-1} - 1$ splits, which becomes increasingly computationally infeasible as q grows.

In practice, the Gini index or cross-entropy measures are usually used when growing classification trees because those measures are more likely to produce pure nodes earlier on. As an example consider a two-class problem and a node with 400 cases of each class. In the first split, the left node has 300 of one class 1 and 100 of the class 2 and the right node has 100 of class 1 and 300 of class 2. In the second split, the left child as 200 of class 1 and 400 of class 2, and the right child has 200 of class 1 and 0 of class 2. In case one, using the misclassification cost gives us :

$$i(m) = p_l * i(m_l) + p_r * i(m_r) = 0.5 * 0.25 + 0.5 * 0.25 = 0.25 \tag{8}$$

whereas the Gini index gives us:

$$i(m) = 0.5(1 - 0.75^2 - 0.25^2) + 0.5(1 - 0.75^2 - 0.25^2) = \frac{3}{8} \tag{9}$$

In the second case, the misclassification cost gives us:

$$i(m) = 0.75 * 0.33 + 0.25 * 0 = 0.25 \tag{10}$$

whereas the Gini index gives us:

$$i(m) = 0.75 * (1 - 0.33^2 - 0.67^2) + 0.25 * (1 - 1^2 - 0^2) = \frac{1}{3} \tag{11}$$

Thus while the misclassification costs are the same for both cases, the Gini index has a lower impurity measure for the second case, which is preferable since it contains a pure node.

1.2 Pruning Trees

Clearly if we continue growing the tree until there are N leaves, then the impurity measure of the tree will be 0. However, this tree would likely have a large degree of overfitting, and may not have much predictive power as a result. Therefore, after growing our tree until $i(T) = 0$, we will need to prune a tree by collapsing subtrees of T into the root node of the subtree. When T' is a subtree of T, we denote this by $T' \preccurlyeq T$. In order to prune the tree, we will first introduce the misclassification cost of a tree T.

1.2.1 Misclassification Costs

In some classification problems, misclassifying a point may be more serious in some cases than others. For example, classifying a person as not having heart disease when they do have heart disease would have worse consequences than classifying someone as having heart disease when they do not. Thus we can sometimes include a $K \times K$ loss matrix L, where $L_{kk'}$ is the cost of classifying a class k observation as class k'. Note that $L_{kk} = 0$. With the new loss matrix, we will classify cases in node m to class $k(m) = \arg\min_k \sum_l L_{lk} * \hat{p}_{ml}$. The misclassification error of node m is:

$$r(m) = \sum_{k'} L_{k'k(m)} \hat{p}_{mk'} \tag{12}$$

The total misclassification of a leaf m is

$$R(m) = r(m) * p(m) \tag{13}$$

where $p(m)$ is the probability of being in node m. Therefore the aggregate misclassification cost of the tree T is therefore given by

$$R(T) = \sum_{m \in \widetilde{T}} R(m) \tag{14}$$

After growing a tree, we will then prune our tree by selecting a subtree that minimizes the cost complexity criterion:

$$R_\alpha(T) = R(T) + \alpha|\widetilde{T}| \tag{15}$$

Here $\alpha \geq 0$ is the cost complexity parameter. For each $\alpha \geq 0$, we want to find the subtree of our original tree T that minimizes $R_\alpha(T)$

1.2.2 Obtaining a Tree Sequence

Let T_α be the smallest subtree of T that minimizes the cost complexity criterion with parameter α. It can be shown that for cost complexity parameters α_1, α_2, if $\alpha_1 \leq \alpha_2$, then $T_{\alpha_1} \preccurlyeq T_{\alpha_2}$. Thus if we already have a pruned subtree $T_{\alpha_1}$and we increase the cost complexity parameter, we can prune directly from T_{α_1} and not the original tree T. Thus if we keep pruning our tree, we will get a sequence of subtrees

$$T \succcurlyeq T_0 \succcurlyeq T_{\alpha_1} \succcurlyeq T_{\alpha_2} \succcurlyeq \cdots \succcurlyeq T_{\alpha_n} = \{m_1\} \tag{16}$$

where $0 \leq \alpha_1 \leq \alpha_2 \leq \cdots \leq \alpha_n$, and $\{m_1\}$ is the subtree containing just the root node of T.

Starting with the original tree T, we will first prune the tree to get T_0. Since $\alpha = 0$, we simply want to the smallest subtree that does not increase the misclassification costs. In order to do this, we will look at every node m of the tree T with two terminal nodes as its children, m_l, m_r. If splitting into m_l, m_r does not improve $R(T)$, that is

$$R(m) = R(m_l) + R(m_r) \tag{17}$$

(since a split can never increase the misclassification rate), then if we collapse the leaves m_l, m_r into its parent, we will obtain a smaller tree with the same cost $R_0(T)$. Continuing this process until no leaves satisfy (17) gives us our optimal tree T_0.

Now consider the tree T_0 and a non-terminal node m. If we obtain $T' \preccurlyeq T_0$ by turning m into a leaf, then m contributes $R(m) + \alpha$ to the total cost complexity measure of T'. Now consider $T'' \preccurlyeq T' \preccurlyeq T_0$, where T'' is obtained from T' by removing m. Then

$$R_\alpha(T'') = R_\alpha(T') - R(m) - \alpha \tag{18}$$

On the other hand, we could also have obtained T'' by removing the entire subtree $T(m)$ with m as its root node. In this case we have

$$R_\alpha(T'') = R_\alpha(T_0) - R_\alpha(T(m)) \tag{19}$$

Equating (18) and (19) gives us

$$R_\alpha(T_0) - R_\alpha(T') = R_\alpha(T(m)) - R(m) - \alpha. \tag{20}$$

Now if $R(m) + \alpha = R_\alpha(T(m))$, then $R_\alpha(T_0) = R_\alpha(T')$, so at this value of α, T' is a better tree than T_0 because it is smaller, but they both have the same cost complexity. Note that

$$\begin{align} R(m) + \alpha &= R_\alpha(T(m)) \tag{21}\\ &= R(T(m)) + \alpha|\tilde{T}(m)| \tag{22}\\ \frac{R(m) - R(T(m))}{|\tilde{T}(m)|} &\leq \alpha \tag{23}\end{align}$$

Thus, defining

$$\begin{align} g(m) &= \frac{R(m) - R(T(m))}{|\tilde{T}(m)|} \tag{24}\\ \alpha_1 &= \min_m g(m) \tag{25}\end{align}$$

for all non-terminal nodes m in T_0, we can prune all the nodes m with $g(m) = \alpha$ to obtain α_1 and $T_{\alpha_1} \preccurlyeq T_0$. Then we will continue pruning the steps to obtain the next α_i and the next pruned tree in our sequence until we are left with just the root node.

1.2.3 Choosing the Tree Size using Cross-Validation

Now that we have a sequence of subtrees and a sequence of parameters α_i, we want to choose the best subtree and its corresponding complexity parameter. We will choose the best complexity parameter via cross-validation. Typically a 10-fold cross-validation will be performed.

Firstly, we have a sequence $T \succcurlyeq T_0 \succcurlyeq T_{\alpha_1} \succcurlyeq T_{\alpha_2} \succcurlyeq \cdots \succcurlyeq T_{\alpha_n} = \{m_1\}$ of trees trained on the entire data set $\mathcal{T}$. Now we will randomly split the training data into 10 equally-sized, non-overlapping chunks, $\mathcal{T}_1, \mathcal{T}_2, \cdots, \mathcal{T}_{10}$. This will give us 10 training sets $\mathcal{T}_1^\complement, \mathcal{T}_2^\complement, \cdots, \mathcal{T}_{10}^\complement$ and 10 corresponding independent test sets $\mathcal{T}_1, \mathcal{T}_2, \cdots, \mathcal{T}_{10}$.

Then for each $i = 1, 2, \cdots, 10$, we will grow a tree T^i from the training set $\mathcal{T}_1^\complement$. For each α in our sequence of parameters, we will then find the optimally pruned tree T_α^i and find the cross-validated misclassification cost of the pruned tree when tested against the test set $\mathcal{T}_i$. For each parameter α, we will add up the misclassification costs from the trees for each $i = 1, 2, \cdots, 10$. The optimal parameter will then be the value of α that minimizes the sum of the cross-validated misclassification costs.

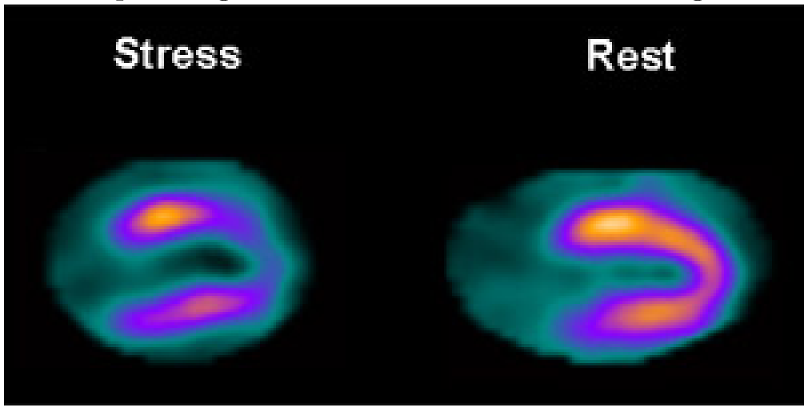
The final tree chosen will therefore be T_α, the tree in our tree sequence trained for the entire training data set, pruned with the optimal parameter α.

2 Numerical Example

2.1 Dataset Description

To illustrate the use of classification trees, we have used our implementation of classification trees on a data set of cardiac Single Proton Emission Computed Tomography (SPECT) images, which is used to detect myocardial perfusion. The images have been classified into one of two classes: 1 for normal and 0 for abnormal. In the original paper presenting this data set, patients were injected with a radioactive tracer and have two three-dimensional images taken- a stress and a rest image. Different slices of the images were then chosen and further subdivided into a total of 22 regions of interest. From each region, the radioactive counts are measured by analyzing the pixels in that region. Thus the feature space has a total of 44 variables- corresponding to 22 regions of interest each for the stress and rest images. For example, "F1S" denotes the count in the first region of interest in the stress image, and "F1R" denotes the count in the first region of interest in the rest image. The entire data set has 267 instances, which the authors of the paper divided into a training set with 80 instances and a test set of 167 instances.

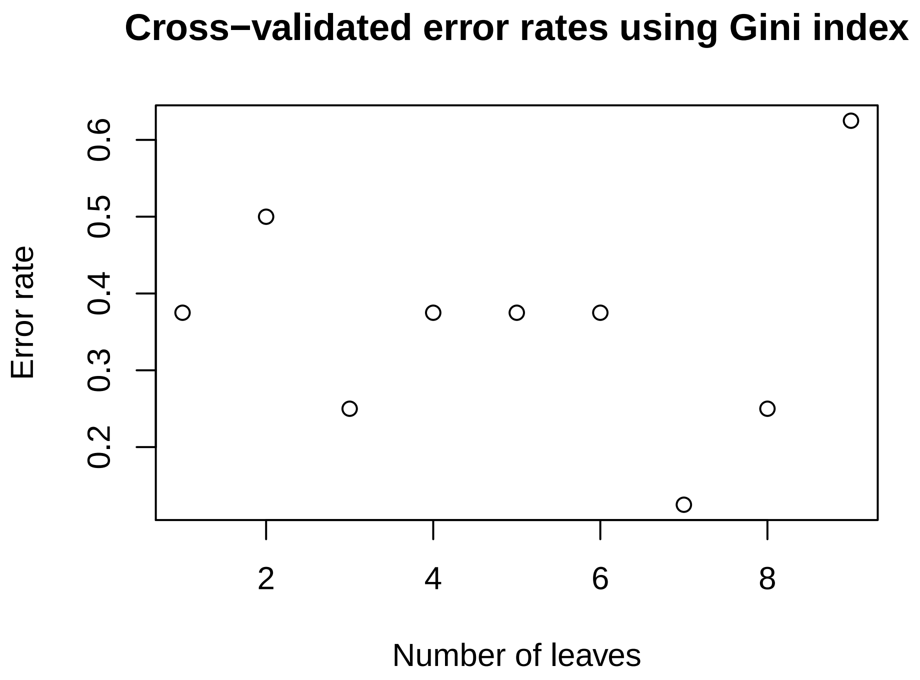
Figure 1: Sample image of a stress and rest SPECT image. Source: [4]



In the original paper [2], the authors analyzed the SPECT imaging data by using an algorithm CLIP3, which is a hybrid of decision tree and rule-based algorithms. While we will not discuss this algorithm in detail, the key differences between CLIP3 and classification trees is that CLIP3 generates partial diagnoses by comparing the values between different variables in the feature space. Once partial diagnoses are obtained, a final diagnosis is reached by taking the conjunction of different partial diagnoses. Thus while classification trees use binary splits on a single variable, CLIP3 generates complex rules comparing the values across different variables. One advantage of classification trees is that they are significantly simpler than CLIP3 and thus possibly easier to interpret. We will therefore choose a decision tree trained on the data and compare our results to the study's result.

2.2 Results

Using the methodology described above, we find the optimal classification tree using two different impurity measures: the Gini index and cross-entropy. For the Gini index, a final tree with 7 leaves was produced, using cross-validation. When tested against the test data, this tree has an error rate of 29.41% A complexity parameter of $\alpha = 5.20417 \times 10^{-18}$ was selected.



When using cross-entropy as the impurity measure, a final tree with three leaves was produced, using cross-validation. When tested against the test set, this produced a significantly worse error rate of 39.04%. The final complexity parameter produced was $\alpha = 0.05$.

Figure 2: Final tree selected via cross-validation using Gini index

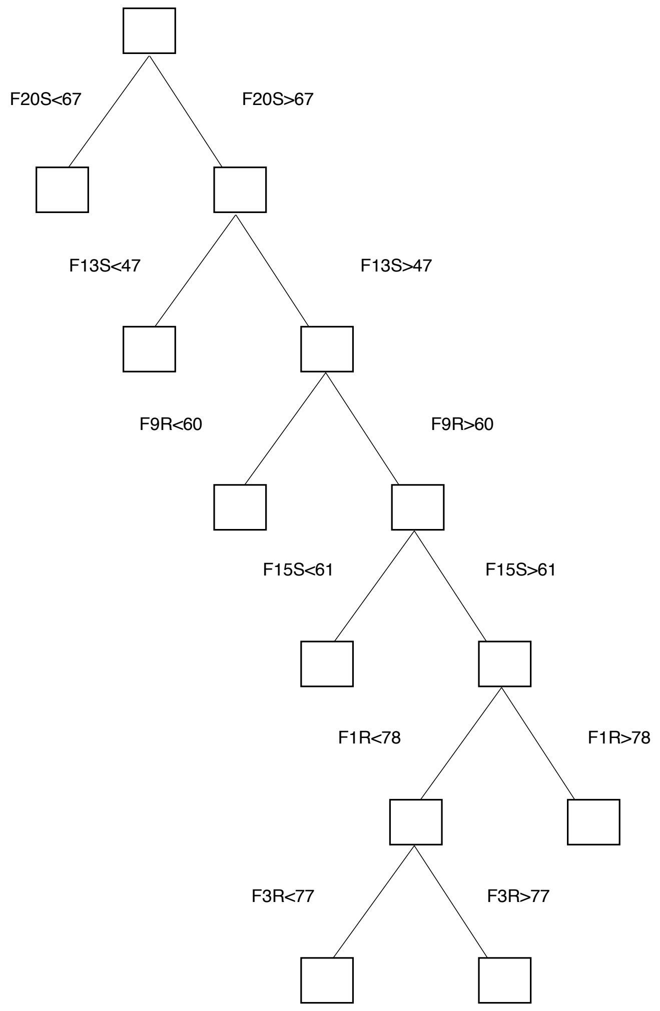
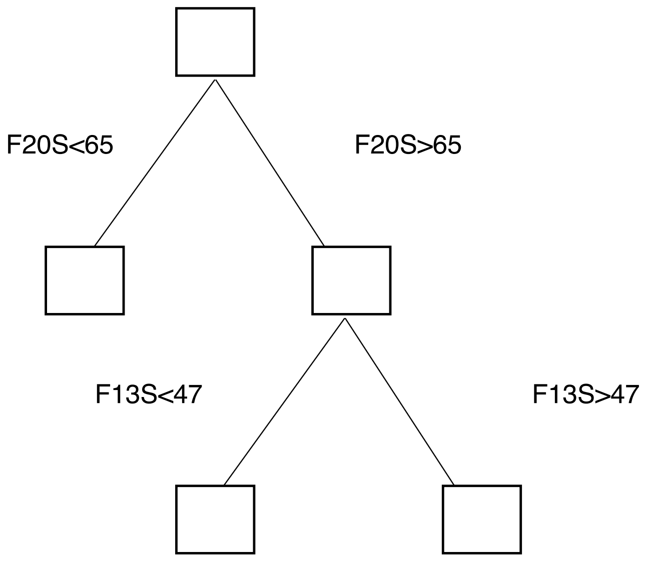
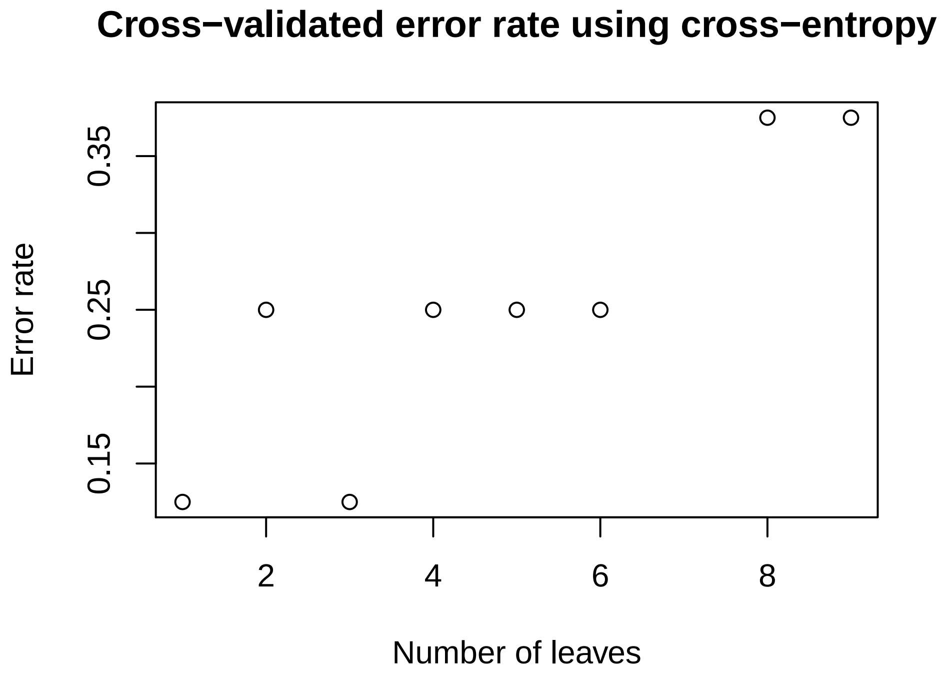


Figure 3: Final tree selected via cross-validation using cross-entropy





By comparison, the algorithm used in the original study produces an error rate of 18.67%, which is significantly better than both of the trees produced in our implementation. However, since CLIP3 is significantly more complex than classification trees, the classification trees still offer a decision-making process that can be helpful. The results could also be further strengthened using bagging or boosting methods, which we have not implemented here.

3 Further Issues

3.1 Instability

One potential issue with decision trees is that they are not very robust to changes in the data. In many cases, a small change to the training data can result in a radically different tree. The reason for this instability is the hierarchical nature of decision trees, as an error in a top level split will affect all proceeding splits below it. One way to mitigate this problem is through bagging- by averaging over many trees, we can reduce the variance.

3.2 Difficulty in Modeling Additive Structures

Another limitation of classification trees is that they do not easily model additive structures of the form $Y_i = \sum_j C_j I(X_j < 2)$. Since the classification trees of binary splits at each level, in order to effectively capture an additive structure, the tree would need to have multiple splits- one for each variable. As the number of variables increases, this leads to an exponential growth in the number of nodes, which makes the tree increasingly hard to interpret. In the numerical example presented above, a classification tree would require many splits to express the types of rules generated by CLIP3. One alternative that mitigates this problem is through Multi-Adaptive Regression Splines (MARS).

References

[1] Trevor Hastie, Robert Tibshirani, Jerome Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction.* New York, NY USA: Springer New York Inc., 2009.

[2] Lukasz A. Kurgan, Krzysztof J. Cios, Ryszard Tadeusiewicz, Marek Ogiela, Lucy Goodenday. *Knowledge discovery approach to automated cardiac SPECT diagnosis* Artificial Intelligence in Medicine, vol. 23/2, pp.149-169, Oct 2001.

[3] David Austin. *How to Grow and Prune a Classification Tree,* `http://www.ams.org/samplings/feature-column/fc-2014-12`

[4] Patient Education - Myocardial Perfusion SPECT, `http://www.brighamandwomens.org/Departments_and_Services/radiology/services/nuclear`

4 Appendix

4.1 Load data

```
#Number of total classes
cnames=c("y")
for(i in 1:22){
    cnames=append(cnames, paste("F",i,"R",sep=""))
    cnames=append(cnames, paste("F",i,"S",sep=""))
}

traindata=read.table("http://archive.ics.uci.edu/ml/machine-learning-databases/spect/
    SPECTF.train", sep=",", col.names=cnames)
K=length(unique(traindata$y)) #Number of classes in data.
J=ncol(traindata)-1
rnum=c(1:nrow(traindata))
traindata=cbind(traindata,rnum)
```

4.2 Impurity measure functions and helper functions

```
#Calculate proportion of class k observations in node m
#m=data in region m
proportion=function(m,k){
    if (nrow(m)==0){
        0
    }else{
        ys=m$y
        N=nrow(m)
        sum(ys==k)/N}
}

#Find class with highest proportion
#Returns both the class and the proportion
classify=function(m){
    p=sapply(c(0:(K-1)),function(x){proportion(m,x)})
    ind=which.max(p)
    c(ind-1,p[ind])
}

#Misclassification error
r=function(m){
    if (nrow(m)==0){
        0
    }else{
        prob=classify(m)[2]
        1-prob}
}

#Gini index
gini=function(m){
    if (nrow(m)==0){
     0
     }else{
        x=sapply(c(0:(K-1)),function(x){proportion(m,x)*(1-proportion(m,x))})
        sum(x)}
}
```

```
#Cross-entropy
centropy=function(m){
    if (nrow(m)==0){
        0
    }else {
    x=sapply(c(0:(K-1)),function(x){proportion(m,x)[2]*log(proportion(m,x)[2])})
    -sum(x)}
}

#Calculate impurity of node m splitting at variable j at point s with impurity
    function fn
#j=feature number, s=split value, m=node, fn=impurity function
#impurity may be gini or cross-entropy

impurity=function(m,j,s,fn){
    N=nrow(m)
    part=partition(m,j,s)
    nl=nrow(part[[1]])
    nr=N-nl
    il=fn(part[[1]])
    ir=fn(part[[2]])

    (nl/N)*il+(nr/N)*ir
}
```

4.3 Building a classification tree recursively

```
#returns list two datasets- left child, right child
#split data on variable j at value s

partition=function(m,j,s){
    l=subset(m,m[,j+1]<=s)
    r=subset(m,m[,j+1]>s)
    list(l,r)
}

#Find best split for a node m
#output = (j,s)
findsplit=function(m,fn){
    N=nrow(m)
    x=matrix(nrow=2,ncol=J)
    for (j in 1:J){
        S=unique(traindata[,j+1])
  i=sapply(S,function(s){impurity(m,j,s,fn)})
        x[,j]=c(min(i),S[which.min(i)])  #min impurity and splitting value for
            variable j
    }
    # x1j=min impurity for variable j
    # x2j=best splitting value for variable j

    jopt=which.min(x[1,])
    sopt=x[2,jopt]

    c(jopt,sopt)
}

#Build a classification tree
#tree= (left, right, j, s, k, r, p)
#j=component, s=splitting value, k=class, r= misclassification cost, p=prob of node
```

```
buildtree=function(initialdata,currentdata,fn){
    T=vector("list",7) #initialize tree
    T[[5]]=classify(currentdata)[1]
    T[[6]]=r(currentdata)
    T[[7]]=nrow(currentdata)/nrow(initialdata)

    if (nrow(currentdata) != 1 & fn(currentdata) != 0) {
        nextsplit=findsplit(currentdata,fn)

  j=nextsplit[1]
  s=nextsplit[2]

  T[[3]]=j
  T[[4]]=s

  children=partition(currentdata,j,s)

  T[[1]]=buildtree(initialdata,children[[1]],fn)
  T[[2]]=buildtree(initialdata,children[[2]],fn)
    }

    T
}
```

4.4 Predicting class of test point

```
#Predict yhat given x
predict=function(tree,x){
    if(! is.leaf(tree)) { #if we are not at a leaf, split accordingly
        j=tree[[3]]
  s=tree[[4]]

        if (x[j+1]<=s) {
      predict(tree[[1]],x)
  } else {
      predict(tree[[2]],x)
  }
    } else {
  tree[[5]]
    }
}
tree.error=function(tree,data){
    n=nrow(data)
    #for each data point, predict y from the tree
    x=sapply(c(1:n),function(i){ predict(tree,data[i,])==data[i,1]})
    #error rate=number of falses / total number data points
    sum(x==FALSE)/n
}
```

4.5 Pruning the tree

```

#Prune initial tree Tmax to get T1 by eliminating all nodes that do not improve
    misclassification
```

```
T_0=function(tree){
    if(is.leaf(tree)){
        tree
    } else if(is.leaf(tree[[1]]) && is.leaf(tree[[2]])){
            r=tree[[6]]*tree[[7]]
      r1=tree[[1]][[6]]*tree[[1]][[7]]
            r2=tree[[2]][[6]]*tree[[2]][[7]]

            if (r1+r2<=r){ #prune by turning parent into leaf
          newleaf(tree[[5]],tree[[6]],tree[[7]])

            } else{ #do not prune if r1+r2>r
          tree
    }
    }else{
            tree[[1]]=T1(tree[[1]])
            tree[[2]]=T1(tree[[2]])
            tree
    }
}

#Get sequence of prune trees corresponding to sequence of alphas
tree.sequence=function(tree){
    t1=T_0(tree)
    T=list(t1)
    as=c(0)
    k=1
    while (numleaves(T[[k]])>1){
        a=findmin(T[[k]])
        as=append(as,a)
        T[[k+1]]=prune(T[[k]],a)
        k=k+1
  print(k)
    }
    list("trees"=T, "alphas"=as)
}

#Prune tree according to parameter a
prune=function(tree,a){
    g=g(tree)
    if (is.leaf(tree)){
        tree
    }else if (g==a){ #prune descendants if g(tree)=a
        newleaf(tree[[5]],tree[[6]],tree[[7]])
    }else{
        tree[[1]]=prune(left(tree),a)
  tree[[2]]=prune(right(tree),a)
  tree
    }
}

left = function (t) { t[[1]] }
right = function (t) { t[[2]] }

#g(t) function
g = function(t) {
    if (numleaves(t)==1){
       Inf
    }else{
        (R(t) - Rsubtree(t)) / (numleaves(t) - 1)
    }
}

#Find minimum value of g(t) of subtree t
```

```
findmin = function(t) {
  if (is.leaf(t)) {
    Inf

  } else {
    me=g(t)
    l=findmin(left(t))
    r=findmin(right(t))

    min(min(me,l),r)
  }
}

#R of a subtree. Used in function g.
Rsubtree=function(tree){
    if (is.leaf(tree)){
        R(tree)
    }else{
        Rsubtree(tree[[1]])+Rsubtree(tree[[2]])
    }
}

#R of a node. Used in function g
R=function(node){ node[[6]]*node[[7]] }

complexity=function(tree,a){ Rsubtree(tree)+a*numleaves(tree) }
```

4.6 Cross-validation functions

```

#Split data into n equal chunks randomly, to use for n-fold cross-validation
splitdata=function(data,n){
    size=floor(nrow(data)/n)
    split(data,sample(rep(1:n,size)))
}

#Get training sets from the splits
trainset=function(data,splits){
    sets=list()
    for (i in 1:length(splits)){
        r=splits[[i]]$rnum
        s=subset(data,!(data$rnum %in% r))
  sets[[i]]=s
    }
    sets
}
```

4.7 Helper functions

```
numleaves=function(tree){
    if (is.leaf(tree)) {
        1
    } else{
        numleaves(tree[[1]])+numleaves(tree[[2]])
```

```
  }
}

is.leaf=function(tree){
    is.null(tree[[3]])
}
newleaf=function(k,r,p){
    l=vector("list",7)
    l[[5]]=k
    l[[6]]=r
    l[[7]]=p
    return (l)
}
```

4.8 Grow trees

```
#Pick final tree using cross-validation
cv.trees=function(data,fn){
    tree=buildtree(data,data,fn)
    seq=tree.sequence(tree) #sequence of pruned trees trained with entire data
    testsets=splitdata(data,10) #split data into 10 sets
    trainsets=trainset(data,testsets)
    alphas=seq$alphas
    trees=seq$trees
    error=rep(0,length(alphas))

    for (j in 1:length(alphas)){ #compute error rate for each test set and each alpha
        trained.tree=buildtree(trainsets[[j]],trainsets[[j]],fn)
  pruned.tree=prune(trained.tree,alphas[j])
  e=0
        for (i in 1:10){
      test.data=testsets[[i]]
      e=e+tree.error(pruned.tree,test.data)
  }
  error[j]=e
    }
    #The best tree is the one with the lowest error rate
    list("alphas"=alphas, "trees"=trees, "error"=error)
}
gini.trees=cv.trees(traindata,gini)
gini.l=sapply(c(1:length(gini.trees$alphas)),function(x){numleaves(gini.trees$trees[[
    x]])})

plot(gini.l, gini.trees$error, main="Cross-validated error rate for different sized
    trees using Gini index", xlab="Number of leaves", ylab="Error rate")
gini.i=which.min(gini.trees$error)
#print(tree.error(gini.trees$trees[[gini.i]],testdata) #Error rate of tree when
    tested against test data


centropy.trees=cv.trees(traindata,centropy)
centropy.l=sapply(c(1:length(centropy.trees$alphas)),function(x){numleaves(centropy.
    trees$trees[[x]])})
plot(centropy.l, centropy.trees$error, main="Cross-validated error rate for different
     sized trees using Cross-entropy", xlab="Number of leaves", ylab="Error rate")
centropy.i=which.min(centropy.trees$error)
print(tree.error(centropy.trees$trees[[centropy.i]],testdata))
}
```