### README

Heng zhe Duan(hd79), Yu Cheng(yc489)

Usage:

The easiest way to use it is to import this project into Eclipse and run from

there.

Main decisions:

We used a proxy pattern on applying the split criterion, where when we passed

in min\_error and max\_gain criterion, the code to the tree will be reused, and

the tree is built according to given criterion automatically.

We also used a static factory to produce the nodes, so that when we need to

change the type of node produced(different stopping parameter or different

splitting criterion), we could simply change the parameter of that factory,

while all other part of the code remains.

### Criterion.java

**import** java.util.ArrayList;

// Interface of criterion

**public** **interface** Criterion {

// Return a double represents how good the split is.

// Higher the value is, better the split is.

**public** **double** calculateSplitPerf(ArrayList<Entry> lchild, ArrayList<Entry> rchild, ArrayList<Entry> all);

}

### Criterion\_MaxGain.java

**import** java.util.ArrayList;

// The criterion that the split with maximum information gain is selected

**public** **class** Criterion\_MaxGain **implements** Criterion {

**private** **static** **double** *LOG2* = Math.*log*(2);

**private** **double** calculateEntropy(ArrayList<Entry> entries) {

**int** countZero = 0;

**int** countOne = 0;

**for** (Entry e : entries) {

**if** (e.label == 0)

++countZero;

**else**

++countOne;

}

**double** p0 = (**double**)(countZero)/((**double**)entries.size());

**double** p1 = (**double**)(countOne)/((**double**)entries.size());

**double** res0 = (p0 == 0)?0:p0\*Math.*log*(p0);

**double** res1 = (p1 == 0)?0:p1\*Math.*log*(p1);

**return** -(res0+res1)/*LOG2*;

}

@Override

**public** **double** calculateSplitPerf(ArrayList<Entry> lchild,

ArrayList<Entry> rchild, ArrayList<Entry> all) {

**double** entropyBefore = calculateEntropy(all);

**double** entropyAfter = ((**double**)lchild.size()) \* calculateEntropy(lchild)

+ ((**double**)rchild.size()) \* calculateEntropy(rchild);

**return** entropyBefore - (entropyAfter)/(**double**)all.size();

}

}

### Criterion\_MinError.java

**import** java.util.ArrayList;

//The criterion that the split with minimum misclassification error is selected

**public** **class** Criterion\_MinError **implements** Criterion {

@Override

**public** **double** calculateSplitPerf(ArrayList<Entry> lchild,

ArrayList<Entry> rchild, ArrayList<Entry> all) {

**int** type0 = Node.*calculateLabel*(all);

**int** errorCount0 = 0;

**for** (Entry e : all) {

**if** (e.label != type0)

errorCount0++;

}

**int** typel = Node.*calculateLabel*(lchild);

**int** errorCount = 0;

**for** (Entry e : lchild) {

**if** (e.label != typel)

errorCount++;

}

**int** typer = Node.*calculateLabel*(rchild);

**for** (Entry e : rchild) {

**if** (e.label != typer) {

errorCount++;

}

}

**return** errorCount0 - errorCount;

}

}

### DataReader.java

**import** java.io.BufferedReader;

**import** java.io.FileReader;

**import** java.util.ArrayList;

**public** **class** DataReader {

**public** **static** ArrayList<Entry> read(String filepath) **throws** Exception{

BufferedReader input = **new** BufferedReader(**new** FileReader(filepath));

String line = **null**;

String featureStr, labelStr;

**int** index;

Integer label;

ArrayList<Integer> features;

ArrayList<Entry> datalst = **new** ArrayList<Entry>();

**while** ((line = input.readLine()) != **null**) {

line = line.substring(0, line.indexOf('#')).trim();

//find label

index = line.indexOf(' ');

labelStr = line.substring(0, index);

**if** (labelStr.length() == 0)

**throw** **new** Exception ("invalid label");

**if** (labelStr.equals("M"))

label = 0;

**else**

label = 1;

line = line.substring(index+1).trim() + " ";

//find features

features = **new** ArrayList<Integer>(9);

**while** (**true**){

index = line.indexOf(' ');

**if** (index == -1)

**break**;

featureStr = line.substring(0, index);

line = line.substring(index+1);

index = featureStr.indexOf(':');

**if** (index == -1)

**throw** **new** Exception ("invalid feature: missing ':'");;

featureStr = featureStr.substring(index+1);

**if** (featureStr.length() == 0)

**throw** **new** Exception ("invalid feature");

features.add(Integer.*parseInt*(featureStr));

}

Integer[] temparray = **new** Integer[features.size()];

features.toArray(temparray);

datalst.add(**new** Entry(label, temparray));

}

input.close();

**return** datalst;

}

}

### Entry.java

**public** **class** Entry {

**public** **int** label;

**public** Integer[] features;

**public** Entry (**int** label, Integer[] features){

**this**.label = label;

**this**.features = features;

}

**public** **static** **void** sanityCheck (Entry e1, Entry e2) **throws** Exception{

**if** (e1.features.length != e2.features.length)

**throw** **new** Exception("unexpected feature size");

}

}

### Main.java

**import** java.util.ArrayList;

**public** **class** Main {

**public** **static** **void** main(String[] args){

**try** {

///////////////////////////////////////////////////////////////////

//// Data used across various parts.

ArrayList<Entry> bcan\_test = DataReader.*read*("bcan/bcan.test");

ArrayList<Entry> bcan\_train = DataReader.*read*("bcan/bcan.train");

ArrayList<Entry> bcan\_train30 = DataReader.*read*("bcan/bcan.train30");

ArrayList<Entry> bcan\_train90 = DataReader.*read*("bcan/bcan.train90");

ArrayList<Entry> bcan\_validate = DataReader.*read*("bcan/bcan.validate");

**int**[] stoppingParamList = **new** **int**[] {1,5,9,17,25,37,43,49};

///////////////////////////////////////////////////////////////////

// Part a)

System.*out*.printf("\n\nPart a)\n");

NodeFactory.*changeCriterion*(**new** Criterion\_MaxGain());

ArrayList<Integer> numberNodes1 = **new** ArrayList<Integer>();

ArrayList<Double> testingError1 = **new** ArrayList<Double>();

ArrayList<Double> validatingError1 = **new** ArrayList<Double>();

ArrayList<Double> trainingError1 = **new** ArrayList<Double>();

**for** (**int** i : stoppingParamList) {

NodeFactory.*changeStoppingParam*(i);

Tree t1 = **new** Tree();

// Grow tree using bcan\_train90

t1. growTree(bcan\_train90);

//t1.printTree();

numberNodes1.add(t1.nodeCount());

testingError1.add(t1.getError(bcan\_test));

validatingError1.add(t1.getError(bcan\_validate));

trainingError1.add(t1.getError(bcan\_train90));

// Print the testing, validate, and training errors

System.*out*.printf("For stopping param = %d, Testing Error " +

"= %f, Validate Error = %f, Training Error = %f, Number" +

" of nodes = %d\n",

i, t1.getError(bcan\_test), t1.getError(bcan\_validate),

t1.getError(bcan\_train90), t1.nodeCount());

}

System.*out*.println("%% For MATLAB plotting purposes only %%");

String xs1 = "";

String testings1 = "";

String trainings1 = "";

**for** (**int** k = 0; k < numberNodes1.size(); ++k) {

xs1 += (numberNodes1.get(k) + ", ");

testings1 += (testingError1.get(k) + ", ");

trainings1 += (trainingError1.get(k) + ", ");

}

xs1 = xs1.substring(0, xs1.length()-2);

testings1 = testings1.substring(0, testings1.length()-2);

trainings1 = trainings1.substring(0, trainings1.length()-2);

System.*out*.println("figure\nhold on");

System.*out*.println("node\_size = [" + xs1 + "]");

System.*out*.println("testing\_error = [" + testings1 + "]");

System.*out*.println("training\_error = [" + trainings1 + "]");

System.*out*.println("plot(node\_size, testing\_error)\n" +

"plot(node\_size, training\_error)\n" +

"xlabel('node number')\nylabel('error')\n" +

"title('Problem 2a) error vs node')\n" +

"legend('testing','training')\n\n");

System.*out*.println("## For table generation purpose only: copy to excel ##");

System.*out*.println("Table of number of nodes Vs. Validation Error");

System.*out*.printf("Stopping Parameter");

**for** (**int** k = 0; k < numberNodes1.size(); ++k) {

System.*out*.printf("\t%d", stoppingParamList[k]);

}

System.*out*.printf("\n");

System.*out*.printf("Number of Nodes");

**for** (**int** k = 0; k < numberNodes1.size(); ++k) {

System.*out*.printf("\t%d", numberNodes1.get(k));

}

System.*out*.printf("\n");

System.*out*.printf("Validation Set Misclassification");

**for** (**int** k = 0; k < numberNodes1.size(); ++k) {

System.*out*.printf("\t%f", validatingError1.get(k));

}

System.*out*.printf("\n");

///////////////////////////////////////////////////////////////////

// Part b)

System.*out*.printf("\n\nPart b)\n");

NodeFactory.*changeCriterion*(**new** Criterion\_MinError());

ArrayList<Integer> numberNodes2 = **new** ArrayList<Integer>();

ArrayList<Double> testingError2 = **new** ArrayList<Double>();

ArrayList<Double> validatingError2 = **new** ArrayList<Double>();

ArrayList<Double> trainingError2 = **new** ArrayList<Double>();

**for** (**int** i : stoppingParamList) {

NodeFactory.*changeStoppingParam*(i);

Tree t2 = **new** Tree();

// Grow tree using bcan\_train90

t2.growTree(bcan\_train90);

numberNodes2.add(t2.nodeCount());

testingError2.add(t2.getError(bcan\_test));

validatingError2.add(t2.getError(bcan\_validate));

trainingError2.add(t2.getError(bcan\_train90));

// Print the testing, validate, and training errors

System.*out*.printf("For stopping param = %d, Testing Error " +

"= %f, Validate Error = %f, Training Error = %f, Number" +

" of nodes = %d\n",

i, t2.getError(bcan\_test), t2.getError(bcan\_validate),

t2.getError(bcan\_train90), t2.nodeCount());

}

System.*out*.println("%% For MATLAB plotting purposes only %%");

String xs2 = "";

String testings2 = "";

String trainings2 = "";

**for** (**int** k = 0; k < numberNodes2.size(); ++k) {

xs2 += (numberNodes2.get(k) + ", ");

testings2 += (testingError2.get(k) + ", ");

trainings2 += (trainingError2.get(k) + ", ");

}

xs2 = xs2.substring(0, xs2.length()-2);

testings2 = testings2.substring(0, testings2.length()-2);

trainings2 = trainings2.substring(0, trainings2.length()-2);

System.*out*.println("figure\nhold on");

System.*out*.println("node\_size = [" + xs2 + "]");

System.*out*.println("testing\_error = [" + testings2 + "]");

System.*out*.println("training\_error = [" + trainings2 + "]");

System.*out*.println("plot(node\_size, testing\_error)\n" +

"plot(node\_size, training\_error)\n" +

"xlabel('node number')\nylabel('error')\n" +

"title('Problem 2b) error vs node')\n" +

"legend('testing','training')\n\n");

System.*out*.println("## For table generation purpose only: copy to excel ##");

System.*out*.println("Table of number of nodes Vs. Validation Error");

System.*out*.printf("Stopping Parameter");

**for** (**int** k = 0; k < numberNodes2.size(); ++k) {

System.*out*.printf("\t%d", stoppingParamList[k]);

}

System.*out*.printf("\n");

System.*out*.printf("Number of Nodes");

**for** (**int** k = 0; k < numberNodes2.size(); ++k) {

System.*out*.printf("\t%d", numberNodes2.get(k));

}

System.*out*.printf("\n");

System.*out*.printf("Validation Set Misclassification");

**for** (**int** k = 0; k < numberNodes2.size(); ++k) {

System.*out*.printf("\t%f", validatingError2.get(k));

}

System.*out*.printf("\n");

///////////////////////////////////////////////////////////////////

// Part c)

System.*out*.printf("\n\nPart c)\n");

NodeFactory.*changeCriterion*(**new** Criterion\_MaxGain());

NodeFactory.*changeStoppingParam*(1);

// Grow tree using bcan\_train90

Tree t3 = **new** Tree();

t3.growTree(bcan\_train90);

// Prune the tree using bcan\_validate

t3.Prune(bcan\_validate);

// Print the test errors, and total number of nodes

System.*out*.printf("The Test Error = %f\nTotal number of nodes = %d",

t3.getError(bcan\_test), t3.nodeCount());

///////////////////////////////////////////////////////////////////

// Part d)

// Please refer to the function.

System.*out*.printf("\n\nPart d)\n");

*crossValidation*(bcan\_train, bcan\_test, stoppingParamList);

///////////////////////////////////////////////////////////////////

// Part e)

System.*out*.printf("\n\nPart e)\n");

NodeFactory.*changeCriterion*(**new** Criterion\_MaxGain());

ArrayList<Integer> numberNodes5 = **new** ArrayList<Integer>();

ArrayList<Double> testingError5 = **new** ArrayList<Double>();

ArrayList<Double> validatingError5 = **new** ArrayList<Double>();

ArrayList<Double> trainingError5 = **new** ArrayList<Double>();

**for** (**int** i : stoppingParamList) {

NodeFactory.*changeStoppingParam*(i);

Tree t5 = **new** Tree();

// Grow tree using bcan\_train30

t5.growTree(bcan\_train30);

numberNodes5.add(t5.nodeCount());

testingError5.add(t5.getError(bcan\_test));

validatingError5.add(t5.getError(bcan\_validate));

trainingError5.add(t5.getError(bcan\_train90));

// Print the testing, validate, and training errors

System.*out*.printf("For stopping param = %d, Testing Error " +

"= %f, Validate Error = %f, Training Error = %f\n",

i, t5.getError(bcan\_test), t5.getError(bcan\_validate),

t5.getError(bcan\_train30));

}

System.*out*.println("%% For MATLAB plotting purposes only %%");

String xs5 = "";

String testings5 = "";

**for** (**int** k = 0; k < numberNodes5.size(); ++k) {

xs5 += (numberNodes5.get(k) + ", ");

testings5 += (testingError5.get(k) + ", ");

}

xs5 = xs5.substring(0, xs5.length()-2);

testings5 = testings5.substring(0, testings5.length()-2);

System.*out*.println("figure\nhold on");

System.*out*.println("node\_size = [" + xs5 + "]");

System.*out*.println("testing\_error = [" + testings5 + "]");

System.*out*.println("plot(node\_size, testing\_error)\n" +

"xlabel('node number')\nylabel('error')\n" +

"title('Problem 2e) training error vs node')\n\n");

} **catch** (Exception e) {

System.*out*.println(e.getMessage());

e.printStackTrace();

}

}

**private** **static** **void** crossValidation(ArrayList<Entry> bcan\_train,

ArrayList<Entry> bcan\_test, **int**[] stoppingParamList) **throws** Exception {

NodeFactory.*changeCriterion*(**new** Criterion\_MaxGain());

// Make a copy

ArrayList<Entry> bcan\_train\_cp = **new** ArrayList<Entry>(bcan\_train);

// Knuth Shuffle.

**for** (**int** i = 0; i < bcan\_train\_cp.size(); ++i) {

**int** loc = MathUtilities.*genRandomIntRange*(i, bcan\_train\_cp.size());

**if** (loc != i) {

Entry e = bcan\_train\_cp.get(i);

bcan\_train\_cp.set(i, bcan\_train\_cp.get(loc));

bcan\_train\_cp.set(loc, e);

}

}

// Generate errors for all cases

**double**[][] bestStoppingParamList = **new** **double**[10][stoppingParamList.length];

**for** (**int** i = 0; i < 10; ++i) {

ArrayList<Entry> training\_p4 = **new** ArrayList<Entry>();

ArrayList<Entry> validating\_p4 = **new** ArrayList<Entry>();

// Build training and validating set with train

**for** (**int** j = 0; j < i\*55; ++j) {

**if** (j < i\*55 || j >= (i+1)\*55)

training\_p4.add(bcan\_train\_cp.get(j));

**else**

validating\_p4.add(bcan\_train\_cp.get(j));

}

**for** (**int** j= 0; j < stoppingParamList.length; ++j) {

NodeFactory.*changeStoppingParam*(stoppingParamList[j]);

Tree t4 = **new** Tree();

// Grow tree using training\_p4

t4.growTree(training\_p4);

// Validate the tree using validating\_p4

// Record the error at bestStoppingParamList[i][j]

bestStoppingParamList[i][j] = t4.getError(validating\_p4);

}

}

**int** bestStoppingParam = -1;

**int** bestError = Integer.*MAX\_VALUE*;

**for** (**int** j = 0; j < stoppingParamList.length; ++j) {

**int** currError = 0;

**for** (**int** i = 0; i < 10; ++i) {

currError += bestStoppingParamList[i][j];

}

**if** (currError < bestError) {

bestError = currError;

bestStoppingParam = stoppingParamList[j];

}

}

// Report bestStoppingParam as best stopping param.

System.*out*.printf("The best stopping parameter = %d\n", bestStoppingParam);

NodeFactory.*changeStoppingParam*(bestStoppingParam);

Tree t4best = **new** Tree();

// Grow tree using bcan\_train

t4best.growTree(bcan\_train);

// Print the testing errors

System.*out*.printf("The Test Error = %f\n",

t4best.getError(bcan\_test));

}

}

### MathUtilities.java

**import** java.util.ArrayList;

**import** java.util.Random;

**public** **class** MathUtilities {

//find euclidean distance between two entries

**public** **static** **double** euc\_dist (Entry e1, Entry e2) **throws** Exception{

Entry.*sanityCheck*(e1, e2);

**double** sum = 0;

**for** (**int** i = 0; i < e1.features.length; i++){

sum += (Math.*pow*(e1.features[i] - e2.features[i], 2.0));

}

**return** Math.*sqrt*(sum);

}

**public** **static** **double** get\_average (ArrayList<Double> lst){

**double** sum = 0;

**for** (Double d : lst){

sum += d;

}

**return** sum/lst.size();

}

**public** **static** **double** get\_stddev (ArrayList<Double> lst){

**double** sum = 0;

**double** average = *get\_average*(lst);

**for** (Double d : lst){

sum += (d - average) \* (d-average);

}

**return** Math.*sqrt*(sum/lst.size());

}

// Compares two doubles to see if they're equal

**public** **boolean** doubleEquals (**double** A, **double** B){

**if** (A == B)

**return** **true**;

**double** maxRelativeError = 0.0001;

**double** relativeError;

**if** (B > A)

relativeError = (B - A) / B;

**else**

relativeError = (A - B) / A;

**return** relativeError <= maxRelativeError;

}

// Generate a random integer in range [lowRange, highRange)

**public** **static** **int** genRandomIntRange(**int** lowRange, **int** highRange) {

Random generator = **new** Random();

**return** lowRange+generator.nextInt(highRange-lowRange);

}

}

### Node.java

**import** java.util.ArrayList;

**import** java.util.HashSet;

**public** **class** Node {

**public** Criterion m\_splitting\_criterion;

**public** ArrayList<Entry> m\_entries;

**public** Node m\_lchild;

**public** Node m\_rchild;

**public** **int** m\_label;

**public** **int** m\_stoppingParam;

**public** **int** m\_splitting\_feature;

**public** **int** m\_splitting\_value;

**public** **boolean** p\_leafNode;

**private** **class** SplitInfo {

**public** ArrayList<Entry> leftLst, rightLst;

//split entrylst into two arraylists<Entry> according to the clause

// left: Entry.attribute.value <= compareValue

// right: Entry.attribute.value > compareValue

**public** SplitInfo (ArrayList<Entry> entrylst, **int** attribute, **int** compareValue) {

**this**.leftLst = **new** ArrayList<Entry>();

**this**.rightLst = **new** ArrayList<Entry>();

**for** (Entry e : entrylst) {

**if** (e.features[attribute] > compareValue)

rightLst.add(e);

**else**

leftLst.add(e);

}

}

}

**public** Node(Criterion criterion, **int** stoppingParam) **throws** Exception {

m\_splitting\_criterion = criterion;

m\_stoppingParam = stoppingParam;

**if** (stoppingParam < 1){

System.*out*.println("stoppingParam: " + stoppingParam);

**throw** **new** Exception ("stoppingParam must be positive");

}

**this**.p\_leafNode = **false**;

}

//gets all possible values for attribute

**private** HashSet<Integer> getAllValues(**int** attribute){

HashSet<Integer> values = **new** HashSet<Integer>();

**for** (Entry e : m\_entries) {

values.add(e.features[attribute]);

}

//take out the largest value

Integer bestI = -1;

**for** (Integer i : values){

**if** (i > bestI){

bestI = i;

}

}

values.remove(bestI);

**return** values;

}

// Split the current tree into left/right children

// Set the splitting feature and values and create l/r child

**public** **void** split() **throws** Exception {

//Stop at the stopping parameter

**if** (m\_entries.size() < m\_stoppingParam){

//stop splitting

m\_label = *calculateLabel*(m\_entries);

p\_leafNode = **true**;

**return**;

}

//stop if no info gain possible (everything has the same label)

**boolean** stop = **true**;

**int** firstLabel = m\_entries.get(0).label;

**for** (Entry e : m\_entries){

**if** (e.label != firstLabel){

stop = **false**;

**break**;

}

}

**if** (stop){

m\_label = firstLabel;

p\_leafNode = **true**;

**return**;

}

//find the local best split

**int** bestAttr = -1;

**int** bestValue = Integer.*MIN\_VALUE*;

**double** bestPerf = -1 \* Double.*MAX\_VALUE*;

ArrayList<Entry> bestLchild = **null**, bestRchild = **null**;

**for** (**int** attr = 0; attr < m\_entries.get(0).features.length; attr++){

HashSet<Integer> attr\_values = getAllValues(attr);

**for** (Integer v : attr\_values){

SplitInfo si = **new** SplitInfo(m\_entries, attr, v);

**double** perf = m\_splitting\_criterion.calculateSplitPerf(si.leftLst, si.rightLst, m\_entries);

//System.out.println("perf: " + perf);

**if** (perf > bestPerf) {

bestLchild = si.leftLst;

bestRchild = si.rightLst;

bestAttr = attr;

bestValue = v;

bestPerf = perf;

}

}

}

**if** (bestPerf <= 0){

//stop splitting

m\_label = *calculateLabel*(m\_entries);

p\_leafNode = **true**;

**return**;

}

//set everything for this node

m\_splitting\_feature = bestAttr;

m\_splitting\_value = bestValue;

m\_lchild = NodeFactory.*returnNode*();

m\_lchild.m\_entries = bestLchild;

m\_rchild = NodeFactory.*returnNode*();

m\_rchild.m\_entries = bestRchild;

m\_label = *calculateLabel*(m\_entries);

p\_leafNode = **false**;

//recursively set everything after;

m\_lchild.split();

m\_rchild.split();

}

/// Find the label of this node(i.e., find all children/decedents

// of this node, and set label to the majority.

**public** **static** **int** calculateLabel(ArrayList<Entry> entrylst) {

**double** sum = 0.;

**for** (Entry e : entrylst) {

sum += e.label;

}

**return** sum/entrylst.size() < 0.5 ? 0 : 1;

}

// The count of nodes in the subtree of this node.

**public** **int** nodeCount(){

**if** (**this**.m\_lchild == **null** && **this**.m\_rchild == **null**)

**return** 1;

**return** 1 + **this**.m\_lchild.nodeCount() + **this**.m\_rchild.nodeCount();

}

}

### NodeFactory.java

// Static factory to generate nodes with given stopping parameter and criterion.

**public** **class** NodeFactory {

**public** **static** Criterion *m\_currCriterion* = **null**;

**public** **static** **int** *m\_currStoppingParam* = -1;

**public** **static** **void** changeCriterion(Criterion criterion) {

*m\_currCriterion* = criterion;

}

**public** **static** **void** changeStoppingParam(**int** stoppingParam) {

*m\_currStoppingParam* = stoppingParam;

}

**public** **static** Node returnNode() **throws** Exception {

**if** (*m\_currCriterion* == **null** || *m\_currStoppingParam* == -1)

**throw** **new** Exception("Need to initalize factory");

**return** **new** Node(*m\_currCriterion*, *m\_currStoppingParam*);

}

}

### Tree.java

**import** java.util.ArrayList;

**public** **class** Tree {

**public** Node root;

**public** **void** growTree(ArrayList<Entry> data) **throws** Exception {

root = NodeFactory.*returnNode*();

root.m\_entries = data;

root.split();

}

**public** **int** nodeCount() {

**return** root.nodeCount();

}

//predicts the label of a predictee using the tree

**public** **int** predictLabel(Entry predictee){

Node currentNode = root;

**while** (!currentNode.p\_leafNode) {

**int** splitting\_feature = currentNode.m\_splitting\_feature;

**int** splitting\_value = currentNode.m\_splitting\_value;

**if** (predictee.features[splitting\_feature] <= splitting\_value)

currentNode = currentNode.m\_lchild;

**else**

currentNode = currentNode.m\_rchild;

}

**return** currentNode.m\_label;

}

//count how many errors are made using a set of entries (this could be the validation set or test set)

**public** **double** getError(ArrayList<Entry> testLst){

**double** errorNumber = 0;

**for** (Entry e : testLst){

**if** (predictLabel(e) != e.label)

errorNumber++;

}

**return** errorNumber/testLst.size();

}

// private class that holds the pair of Node and the error if pruning this node.

**private** **class** PruneData {

**public** Node n;

**public** **double** error;

**public** PruneData(Node n, **double** error){

**this**.n = n;

**this**.error = error;

}

}

//prune at the node that minimizes error

**public** **void** Prune(ArrayList<Entry> validationLst) {

**while** (**true**) {

//error without pruning

**double** default\_error = getError(validationLst);

PruneData best = whereToPrune(root, validationLst);

//if root == leaf (edge case)

**if** (best.n == **null**)

**break**;

//best to not prune

**else** **if** (best.error >= default\_error)

**break**;

//prune

**else** {

best.n.m\_lchild = **null**;

best.n.m\_rchild = **null**;

best.n.p\_leafNode = **true**;

}

}

}

//find the best place to prune (node that minimizes error)

**public** PruneData whereToPrune(Node n, ArrayList<Entry> validationLst){

//Stop at leaf node

**if** (n.m\_lchild == **null** && n.m\_rchild == **null**)

**return** **new** PruneData(**null**, Double.*MAX\_VALUE*);

//prune at current node

n.p\_leafNode = **true**;

**double** error\_current = getError(validationLst);

n.p\_leafNode = **false**;

//prune somewhere in left branch

PruneData p\_left = whereToPrune(n.m\_lchild, validationLst);

//prune somewhere in right branch

PruneData p\_right = whereToPrune(n.m\_rchild,validationLst);

//find best place to prune

PruneData best;

**if** (error\_current > p\_left.error){

**if** (p\_left.error > p\_right.error)

best = p\_right;

**else**

best = p\_left;

} **else** {

**if** (error\_current > p\_right.error)

best = p\_right;

**else**

best = **new** PruneData(n, error\_current);

}

**return** best;

}

// Print a node in the tree with given indentation

**private** **void** printNode(Node e, **int** indentation) {

**for** (**int** i = 0; i < indentation; ++i) {

System.*out*.printf("\t");

}

**if** (e.p\_leafNode) {

System.*out*.printf("Classify as %d with [", e.m\_label);

**for** (Entry j : e.m\_entries) {

System.*out*.printf("%d ", j.label);

}

System.*out*.println("]");

}

**else** {

System.*out*.printf("Split on feature %d, at threshold = %d\n",

e.m\_splitting\_feature, e.m\_splitting\_value);

printNode(e.m\_lchild, indentation+1);

printNode(e.m\_rchild, indentation+1);

}

}

// DEBUG ONLY: print the decision tree.

**public** **void** printTree() {

printNode(root, 0);

}

}