CS 540-2: Introduction to Artificial Intelligence

Homework Assignment #3

Assigned: Monday, February 20
Due: Saturday, March 4

Hand-In Instructions

This assignment includes written problems and programming in Java. **The written problems must be done individually but the programming problem can be done either individually or in pairs.** Hand in all parts electronically by copying them to the Moodle dropbox called "`HW3 Hand-In`". Your answers to each written problem should be turned in as separate pdf files called `<wisc NetID>-HW3-P1.pdf` and `<wisc NetID>-HW3-P2.pdf`. If you write out your answers by hand, you'll have to scan your answer and convert the file to pdf. Put your name at the top of the first page in each pdf file. **Both people on a team *must individually answer* and submit their answers to the written problems; no collaboration on these problems is allowed!**

For the programming problem, put all the java files needed to run your program, including ones you wrote, modified or were given and are unchanged (including the **required** `HW3.java`), into a folder called `<wisc NetID>-HW3` Compress this folder to create `<wisc NetID>-HW3-P3.zip` and copy this file to the Moodle dropbox. Make sure your program compiles and runs on CSL machines. Your program will be tested using several test cases of different sizes. Make sure all files are submitted to the Moodle dropbox.

Late Policy

All assignments are due **at 11:59 p.m.** on the due date. One (1) day late, defined as a 24-hour period from the deadline (weekday or weekend), will result in 10% of the total points for the assignment deducted. So, for example, if a 100-point assignment is due on a Wednesday and it is handed in between any time on Thursday, 10 points will be deducted. Two (2) days late, 25% off; three (3) days late, 50% off. No homework can be turned in more than three (3) days late. Written questions and program submission have the same deadline. A total of three (3) free late days may be used throughout the semester without penalty. Assignment grading questions must be raised with the instructor within one week after the assignment is returned.

Problem 1. [16] Unsupervised Learning

Consider the following information about 10 two-dimensional instances (aka examples) that are labeled with their class, T or F:

	X1	X2	Class
x1	0.5	4.5	T
x2	2.2	1.5	T
x3	3.9	3.5	F
x4	2.1	1.9	T
x5	0.5	3.2	F
x6	0.8	4.3	F
x7	2.7	1.1	T
x8	2.5	3.5	F
x9	2.8	3.9	T
x10	0.1	4.1	T

	Nearest C*i* After 1 K-Means Iteration (i.e., C1, C2 or C3)	3 Nearest Neighbors, *xi*, *xj*, *xk*	3-NN Predicted Class	Cluster Majority-Vote Predicted Class
x1				
x2				
x3				
x4				
x5				
x6				
x7				
x8				
x9				
x10				

(a) [6] Perform (manually) **one** (1) iteration of **K-means clustering** using $k = 3$, Euclidean distance, and initial cluster centers C1 = (0.8, 4), C2 = (0.5, 0.5), and C3 = (2, 2).

(i) [4] Give the new positions of C1, C2 and C3 after one iteration.

(ii) [2] In the table above, fill in the first column, labeling each point *xi* with it's nearest cluster center, C1, C2 or C3.

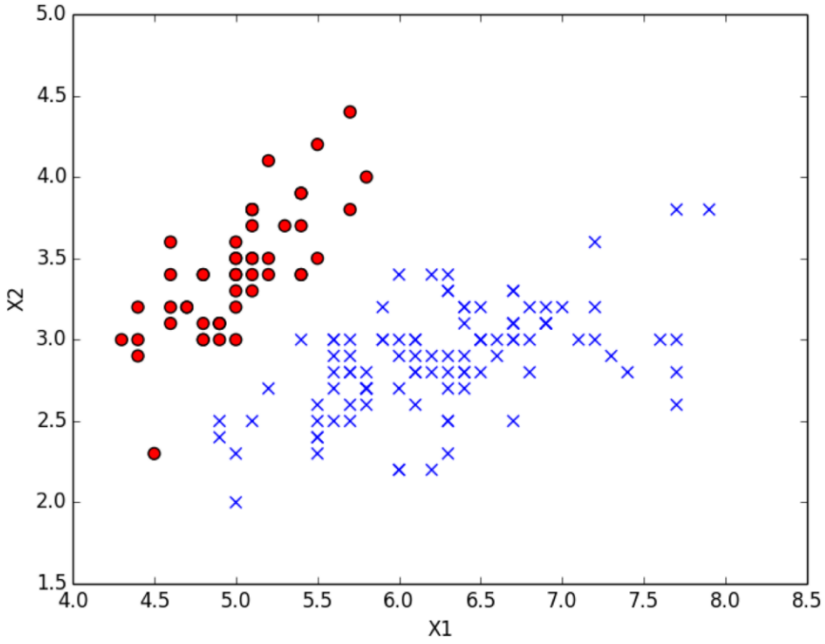
(b) [4] In the table above, fill in the second column, labeling each point *xi* with it's 3 nearest neighbors, excluding the point itself (i.e., don't consider a point as one of its nearest neighbors).

(c) [2] In the table above, fill in the third column, labeling each point *xi* with the majority class label of its 3 nearest neighbors.

(d) [2] In part (a)(ii) you identified the nearest cluster center to each point *xi*. Assign each cluster center C*i* the majority class (T or F) of the points that are nearest to that cluster center (e.g., if 2 of the 3 closest points to C1 had class T, then C1 would be labeled T). In the case where there are an equal number of points belonging to Ci with labels T and F, label the cluster center 'F'. Next, fill in the fourth column, labeling each point *xi* with the class of its nearest cluster center (e.g., if *x*3 was closest to C1 and C1 had majority class T, then *x*3 would be labeled T). Call this classification method "Nearest Cluster Center."

(e) [2] The last two columns of the table on the right in the previous page correspond to predicted labels for each *xi*. Which method, 3-NN or Nearest Cluster Center, assigns the most correct labels (i.e., the predicted class equals the true class)? What is the fraction of the number of correctly labeled points to the total number of points for the better method?

Problem 2. [18] Decision Trees



(a) [4] Manually Construct a Decision Tree

(i) [2] In the figure above, draw a set of split boundaries that could possibly be defined by a decision tree, such that its associated decision boundary would **perfectly** classify all the data points. You can split on attributes X1 and X2 as many times as necessary. (We are not looking for splits based on any split metric such as information gain; rather, we are looking for a high-level understanding of decision trees. Note that there are multiple possible correct answers).

(ii) [2] Draw the decision tree corresponding to the split boundaries you drew in part (a)(i). (At leafs, indicate class labels with either 'O' or 'X'.)

Instance	X1	X2	X3	Class
1	T	T	5.1	Y
2	T	T	7.5	N
3	T	F	8	N
4	F	F	3	N
5	F	T	7	Y
6	F	T	4	Y
7	F	F	5	Y
8	T	F	6	Y
9	F	T	1	Y

(b) [14] Decision Trees and Split Points

(i) [10] Given the 9 instances in the table above, calculate the information gain of each candidate split point for each attribute. Be sure to indicate what attribute each result corresponds to, and the split threshold used for the continuous attribute, X3. Perform splits on X3 that are binary, and only occur at class boundaries (See 'How to split on real-valued attributes' section in the programming problem for more information).

(ii) [2] Based on your results in (b)(i), which attribute will be selected for the root of the decision tree? If two attributes have equal information gain, select the attribute appearing in the left most column of the table above. If two split thresholds for X3 produce the same information gain, select the larger threshold (Include the split threshold if the split attribute selected is X3).

(iii) [2] What would happen if we considered the column labeled "Instance" (where each row has a unique instance number) as a categorical attribute that we could perform a split on (i.e., one split for each category (integer) in the table above)? Do you think this attribute should be used for a decision tree? Explain why or why not. (Hint: Think about predicting new data points.)

Problem 3. [66] Implementing Real-Valued Decision Trees

Overview

In this problem you are to implement a program that builds a binary decision tree for real-valued numerical attributes, and binary classification tasks. By binary tree, we mean that every non-leaf node of your tree will have exactly two children. Each node will have a selected attribute and an associated threshold value. Instances with an attribute value less than or equal to the threshold belong to the left subtree of a node, and instances with an attribute value greater than the threshold belong to the right subtree of a node. The programming part requires building a tree from a training dataset and classifying instances of both the training set and the testing set with the learned decision tree. Code has been provided for printing your decision tree in a specific format. You may change the code if you wish, but the output must be in *exactly* the same format that the function provides. The function we've provided for printing the tree is called `print()` and can be found in `DecisionTreeImpl.`

We have provided to you skeleton code to help you on this problem. While you are not required to use the code, there are some aspects of your submission that **you must conform to**. The requirements of your submitted code are:

1. You must submit at least one java file named `HW3.java`, and it must contain your homework's static main method.
2. Your `HW3.java` file should accept 4 arguments, specified by the command: `java HW3 <modeFlag> <trainFile> <testFile> <k instances per leaf limit>`
3. Your `HW3.java` file should be able to read data in the format we specify, and print the correct output for each of the modes. Your output **must** be **exactly** (Including matching whitespace characters) the same as the example output we have provided to you. Mismatched or mis-formatted output will be marked as incorrect.
4. Any supporting java files that you use with your code must be submitted with your `HW3.java` file.

In addition to constraints on the program files, there are also some simplifying assumptions that you can use when developing your code:

1. All attributes will be real valued. There will be no categorical attributes in the set of training or test instances.
2. Splits on real-valued attributes are binary (i.e., each split creates a <= set and a > set).
3. A real-valued attribute can be split on multiple times in the decision tree.
4. All attributes will appear in the same order for every instance (a row in the data file), and will always be separated by commas only.
5. The last column of every row in the data file will always contain the class label for that specific instance.
6. The class label will only ever be a 1 or 0.

7. Lines starting with "##" indicate an attribute label name, only occur at the beginning of a file, and occur in the same order as the attributes in each row. There is no "##"line for the class label in each row.

Aspects of The Tree

Your decision tree must have certain properties so that the program output based on your decision tree implementation matches our own. The constraints are:

1. The attribute values of your instances and all calculations must be done with doubles.
2. At any non-leaf node in the tree, the left child of a node represents instances that have an attribute value less than or equal to (<=) the threshold specified at the node. The right child of a node represents instances that have an attribute value greater than (>) the threshold specified at the node.
3. When selecting the attribute at a decision tree node, first find the **best** (i.e., highest information gain) split threshold for each attribute by evaluating multiple candidate split thresholds. Once the best candidate split is found for each attribute, choose the attribute that has the highest information gain. If there are multiple attributes with the same information gain, split on the attribute that appears **later** in the list of attribute labels. If an attribute has multiple split thresholds producing the same best information gain, select the threshold with higher value.
4. When considering creating a decision tree node, if the number of instances belonging to that node is less than or equal to the "k instances per leaf limit", a leaf node must be created. The label assigned to the node is the majority label of the instances belonging to that node. If there are an equal number of instances labeled 0 and 1, then the label 1 is assigned to the leaf.
5. When the set of instances at a node are all found to contain the *same* label, splitting should stop and a leaf should be created for the set of instances.

How to Split on Real-Valued Attributes

Splitting on real-valued attributes is slightly different than splitting on categorical attributes. We only ask you to consider binary splits for real-valued attributes. This means that an attribute "threshold" value must be found that can be used to split a set of instances into those with attribute value less than or equal to the threshold, and those with attribute value greater than the threshold. Once instances are separated into two groups based on a candidate threshold value, the information gain for the split can be calculated in the usual way.

The next question is, how do we calculate candidate thresholds to use for an attribute? One good way is to sort the set of instances at the current node by attribute value, and then find consecutive instances that have *different* class labels. A difference in information gain can only occur at these boundaries (as an exercise, think about why this is given the way information gain works). For each pair of consecutive instances that have different classes, compute the average value of the attribute as a candidate threshold value, even if the two values are equal. However, make sure that instances are sorted first by attribute value, and second by class value

in ascending order (Class 0 before Class 1). This results in a much smaller number of splits to consider. An example of the whole process for splitting on real-valued attributes is given below. The steps for determining candidate split thresholds for attribute X2:

1. Sort instances at the current node by the value they have for attribute X2.
2. Find pairs of consecutive instances with different class labels, and define a candidate threshold as the average of these two instances' values for X2.
3. For each candidate threshold, split the data into a set of instances with X2 ≤ threshold, and instances with X2 > threshold.
4. For each split, calculate the information gain of the split.
5. Set the information gain for splitting on attribute X2 as the largest information gain found by splitting over all candidate thresholds.

Set of Instances at Current Node

	X1	X2	Class
x1	0.5	4.5	F
x2	2.2	1.5	T
x3	3.9	3.5	F
x4	2.1	1.9	T
x5	1.1	4	T

Instances After Sorting on Values of X2

	X1	X2	Class
x2	2.2	1.5	T
x4	2.1	1.9	T
x3	3.9	3.5	F
x5	1.1	4	T
x1	0.5	4.5	F

$$\text{Candidate Thresholds} = \{\frac{1.9+3.5}{2} = 2.7, \frac{3.5+4}{2} = 3.75, \frac{4+4.5}{2} = 4.25\}$$

Instances with X2 ≤ 2.7

	X1	X2	Class
x2	2.2	1.5	T
x4	2.1	1.9	T

Instances with X2 > 2.7

	X1	X2	Class
x3	3.9	3.5	F
x5	1.1	4	T
x1	0.5	4.5	F

The information gain after splitting on X2 with threshold 2.7 is 0.9709 - 0 - .5509 = 0.4199
Similarly, information gain is computed for X2 with threshold 4.25, and the threshold producing largest information gain is selected the best threshold value for attribute X2.

Description of Program Modes

We will test your program using several training and testing datasets using the command line format:

`java HW3 <modeFlag> <trainFile> <testFile> <k instances per leaf limit>`
where `trainFile` and `testFile` are the names of the training and testing datasets, formatted to the specifications described later in this document. The `modeFlag` is an integer from 0 to 3, controlling what the program will output. Only `modeFlag` = {0, 1, 2, 3} are required to be implemented for this assignment. The requirements for each value of `modeFlag` are described in the following table:

0: Print the best possible information gain that could be achieved by splitting on each attribute at the root node, based on all the training set data 1: Create a decision tree from the training set, with a limit of *k* instances per leaf, and print the tree 2: Create a decision tree from the training set, with a limit of *k* instances per leaf, and print the classification for each example in the training set, and the accuracy on the training set 3: Create a decision tree from the training set, with a limit of *k* instances per leaf, and print the classification for each example in the test set, and the accuracy on the test set

Suggested Approach
We have provided you with a skeleton framework of java code to help you in working on this project. While you don't have to use the skeleton code, we encourage you to do so, since much of the input and output work that we expect has been completed for you. Feel free to change the classes, function interfaces, and data sets within the skeleton framework as you are allowed to implement the algorithm in whatever way you see fit. The only restrictions are that the program still functions the same as stated earlier, and that the output **exactly** matches the expected output for each mode.

In the skeleton code we provide, there are four main methods you should focus on implementing:

```
1. DecisionTreeImpl();
2. private DecTreeNode buildTree();
3. private double calculateEntropy();
4. public String classify();
5. public void rootInfoGain();
6. public void printAccuracy();
```

1. `DecisionTreeImpl()` initializes your decision tree. You can call the recursive `buildTree()` function here to initialize the root member of `DecisionTreeImpl` to a fully-constructed decision tree.

2. `buildTree()` is a recursive function that can be used to build your decision tree. After each call it returns a `DecTreeNode`.
3. `classify()` predicts the label (0 or 1) for a provided instance, using the already constructed decision tree.
4. `rootInfoGain()` prints the best information gain that could be achieved by splitting on an attribute, for all the attributes at the root (one in each line), using the training set.
5. `printAccuracy()` prints the classification accuracy for the instances in the provided data set (i.e., either the training set or test set) using the learned decision tree.

Data

The example datasets we've provided come from the UW cancer cell database, and can be found at the UCI machine learning repository. Each attribute is some characteristic of a potential cancer cell, and the class labels, 0 and 1, stand for benign and malignant. All data sets used for testing your code will conform to the properties outlined earlier in this document, and are re-specified below:

1. All attributes are real valued, and the last column of each row corresponds to a numeric class label of 0 or 1.
2. Each line starting with "##" corresponds to an attribute label, and occurs in the same order as the attributes in each row. The class label does not have a corresponding "##" prefixed line.
3. Each row corresponds to a single data instance, with all attributes and labels separated by commas only.

For example, the first instance in the data file below corresponds to the feature vector (A1=50, A2=120, A3=244, A4=162, A5=1.1) and its class is 1.

```
// Description of the data set
##A1
##A2
##A3
##A4
##A5
50,120,244,162,1.1,1
40,110,240,160,1,1
62.7,80.3,24,62,2.5,0
…
```

Example Output

Mode 0:

Given the command:

```
java HW3 0 train1.csv test1.csv 10
```

the output should be:

```
A1 0.462986
A2 0.159305
A3 0.466628
…
A30 0.074669
```

Mode 1:
Given the command:

```
java HW3 1 train1.csv test1.csv 10
```

the output should be:

```
A23 <= 105.950000
|    A28 <= 0.135050
|    |    A14 <= 48.975000
|    |    |    A22 <= 30.145000: 0
|    |    |    A22 > 30.145000: 0
|    |    A14 > 48.975000: 1
|    A28 > 0.135050: 1
A23 > 105.950000
|    A23 <= 117.450000: 1
|    A23 > 117.450000: 1
```

Mode 2:
Given the command:

```
java HW3 2 train1.csv test1.csv 10
```

the output should be:

```
1
0
1
…
1
1
0.912389
```

Mode 3:
Given the command:

```
java HW3 3 train1.csv test1.csv 10
```

the output should be:

```
0
0
1
…
1
1
0.87656
```

Final Note On Submission

As part of our testing process, we will unzip the file you submit to Moodle, call `javac *.java` to compile your code, and then call the main method `HW3` with parameters of our choosing. Make sure that your code runs on the computers in the department because we will conduct our tests using these machines.