Programming Assignment 3 - Heap and Decision Tree

Due: March 25

This is an individual assignment!

You are responsible for adhering to the Aggie Honor Code and student rules regarding original work. You must write and submit your own code to Mimir. Plagiarism checks will be used to check the code in addition to a manual review from the grader.

That said, we recognize that there is a certain degree of collaboration and referencing involved in the process of problem solving. In the starter code, there is a provided `bibliography.txt` where you will track all of the references you use. Use references generally to help think through a problem or find information about an error. This is especially true if you talk with peers: focus on the high-level aspects of the problem. *Do not share code! Do not copy code!*

Overview

About this Assignment

We have spent some time talking about tree structures and the types of applications they support. Trees are widely used for algorithms that require logarithmic runtime because they store the data down branches which reduce the problem size into subtrees. In the labs, you have implemented a fully-operational binary search tree, and you have also done several exercises involving balancing trees, recursive operations such as traversals, and practicing the operations for certain special trees such as heaps. While we will discuss many more special trees in upcoming lectures and labs, this assignment focuses on heaps and decision trees.

A **heap** is a tree which is used to implement a priority queue. That is, it stores information in such a way that the maximum or minimum element is always on top of its respective subtree. We have seen examples of min-heaps in class. This assignment requires you to implement a max-heap in order to support the training phase of a decision tree.

A **decision tree** is a special type of binary tree used to make a decision process. Each node of a decision tree is a decision point. Based on some parameter, we either take the flow direction of the left child or the right child. Decision trees are one popular form of supervised machine learning.

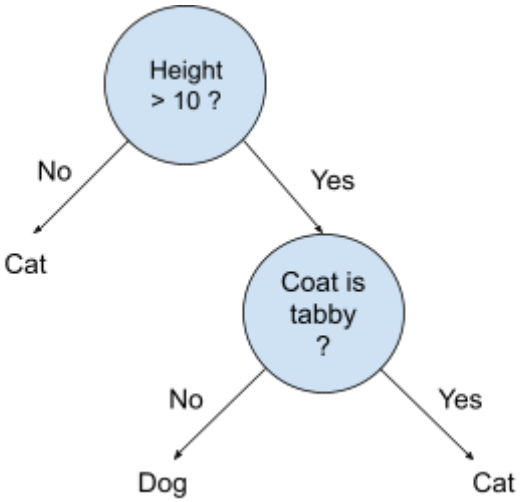
Machine Learning and Decision Trees - A Short Introduction

The general topic of machine learning is outside the scope of this course, but in a nutshell, machine learning is about creating algorithms and structures able to learn from observations and respond "intelligently." Unsupervised machine learning takes unlabeled data and tries to interpret it by grouping similar observations together. Supervised learning involves providing large amounts of labeled data to an algorithm which is able to learn the relationship between the observations and the outcomes. Supervised learning is widely used today to power a variety of tools like speech recognition, computer vision and image processing, natural language processing, and more. Decision trees are a form of supervised learning that examines the available data and creates decision points that help it best categorize the outcomes.

Consider the following simple dataset:

Weight	Height	Coat Color	Breed
8.1 lbs	9.1 in	black	Cat
11 lbs	10.1 in	brown	Dog
10.2 lbs	10.5 in	tabby	Cat
9.2 lbs	8.9 in	white	Cat
10.1 lbs	12 in	black	Dog

We might create a very simple decision tree as follows:



This is a small example of applying decision trees to a binary dataset, that is, a set of data with two possible outcomes. Decision trees can support more outcomes than that, but we will focus on binary problems for this assignment, in particular, data for health-based applications.

Decision trees are fairly unsophisticated by themselves. They simply create a hierarchy of choices that describe the data. However, they can become very powerful when used in combination with each other. This grouping of decision trees into a single model is called a random forest, and it is a common type of machine learning classifier (an algorithm which creates a data model to "classify" or identify a particular outcome, e.g., cat vs. dog).

Decision trees are trained using *information gain*. By looking at the data and considering all possible choices for decision points, the goal is to find a decision that will tell us the most about the problem. In this example, we would want to find an attribute and data point that helps us to separate the most cats from dogs. Height > 10 is an example of a single threshold that can split most data points.

You will not need to code the information gain function for this assignment! It is well beyond the scope of this class, although you can read more about it here. To see a fun discussion and application of information gain, watch 3Blue1Brown's video analyzing Wordle using information theory.

Getting Started

In this assignment, you will first implement an array-based heap. Next, you will implement a decision tree. We will leverage concepts from earlier assignments, as well as structures provided by the standard template library (STL) in C++.

1. Go to the assignment on Mimir and download the starter code.
    a. There is a `bibliography.txt` file where you should track the references you use for this assignment.
    b. The following files are fully implemented and will not need to be edited:
        i. `DecisionLogic.h` - header file for the Decision class and helper functions such as the information gain calculator
        ii. `DecisionLogic.cpp` - source code which implements the helper functions to read the data into parallel vectors (essentially a table) and can calculate information gain for a specific attribute
        iii. `runner.cpp` - a program which loads input data from provided arguments and trains and tests a decision tree
    c. The MaxHeap, DNode, and DTree are unimplemented.
2. There are provided datasets for detecting diabetes and cancer based on a set of readings. You will use these with the runner to test your decision tree.
    a. The diabetes dataset comes from a Coursera class[1] and the cancer dataset is the "Wisconsin Breast Cancer Database" from the University of Wisconsin Hospitals, Madison[2].
    b. As an example, once you have a compiled program, you can run it with the data for cancer detection with the following (assuming you named the program *runner*):

```
./runner train_cancer.csv test_cancer.csv
```

3. Implement the MaxHeap
    a. As we discussed in lecture, heaps are tree-based structures which can be built on top of trees or on top of arrays. While arrays and linear objects are generally more work to use as a basis for generic trees, because of the property that they *must* be complete trees, heaps can be implemented on top of arrays quite easily.
    b. You may use vector for your heap. You may also use your own dynamic array or tree-based implementation. In any case, you should make your heap compatible with the function signatures defined in Part A.

[1] https://github.com/adityakumar529/Coursera_Capstone/blob/master/diabetes.csv
[2] https://archive.ics.uci.edu/ml/machine-learning-databases/breast-cancer-wisconsin/

4. Implement the DNode and DTree

    a. The decision tree will look quite similar to trees we have seen so far, but it will have highly specialized operations given its sole purpose is to serve as a classifier.

    b. The decision tree will need to use the max-heap from the previous step, so you will find it easiest to start with the heap.

We are not requiring a makefile for this assignment since it is straightforward to compile the sources together, unlike in the previous assignment where we had intermediate build operations to generate the library. That said, you are encouraged to use a makefile to practice building them.

Part A: MaxHeap

Task

Implement a templated max-heap in `MaxHeap.h`. Your max-heap will ultimately be used for tracking the maximum information gain used by the decision tree, but we will test it independently as a fully functioning implementation of a generalized heap. Your heap only needs a single template type, not a key-value pair, since we'll be using a separate object to hold the key-value pair together.

You must implement the following functions inside the MaxHeap class. For the purposes of testing, **make each of these functions public**!

MaxHeap Class Function Signatures	Description
`MaxHeap()`	Default constructor
`MaxHeap(vector<T>)`	Constructor accepts a vector of generic objects in any order and converts it into a valid max-heap via heapify
`int getParent(int) const`	Given the index of the current node, returns the index of the parent node
`int getLeft(int) const`	Given the index of the current node, returns the index of the left child node
`int getRight(int) const`	Given the index of the current node, returns the index of the right child node
`bool empty() const`	Boolean indicates if the heap is empty or not
`void upheap(int)`	Given the index of a node, propagate it up the max heap if the parent is a smaller value
`void downheap(int)`	Given the index of a node, propagate it down the max heap if a child has a larger value
`void heapify()`	Heapify the current structure to ensure the rules of a heap are maintained; used in conjunction with the parameterized constructor that accepts a vector

`void insert(T)`	Insert an element into the heap; used with upheap
`T removeMax()`	Remove an element from the heap; returns max element and uses downheap

After completing the MaxHeap class, you should implement the heapsort algorithm by adding the function definition to the bottom of your `MaxHeap.h` file. Note this will be a reverse-sort algorithm where the largest elements are first, since it is based on a max-heap.

MaxHeap.h Function Signatures	Description
`vector<T> heapsort(vector<T> v)`	Receives an unsorted list and returns a reverse-sorted list (largest elements first)

Point Distribution

This part comprises 58% of the overall assignment score. That is distributed as follows:

- 15% for getters and empty() check
- 16% for upheap and downheap
- 12% for constructors and heapify operation
- 10% for insert and remove
- 5% for heapsort

Part B: Decision Tree

Task

Implement the DNode and DTree classes. You do not need to write the runner, and all of the logic operations for reading CSV files into data tables and finding the information gain for each attribute are provided through the DecisionLogic files.

Additionally, these classes do not need to be templated! Although we have been using templating almost everywhere since introducing the concept, this is an example of a class that is so specialized for its data type, that it does not save very much effort to template, at least not during the initial implementation.

Start with the DNode class, which is very small and only needs the following **public** members:

DNode Class Data Member	Description
`Decision key`	The "key" data member will be of type "Decision", included from "DecisionLogic.h". Note the Decision object holds information like the attribute and decision threshold.
`int depth`	A node should know its depth to enforce the maxdepth of

	the tree and to use for printing. Root is at depth = 0; its children at depth = 1; and so on.
`DNode* parent`	Pointer to parent
`DNode* left`	Pointer to left child
`DNode* right`	Pointer to right child

Next, implement the data and function members for the DTree class in `DTree.h`. Again, large amounts of logic have been provided for you through the already-implemented code. However, the DTree class is not without some challenges. It is very similar to the trees we have seen before, including some concepts from labs, but it is slightly unique to this application.

DTree Class Data Member	Description
`DNode* root`	Root of the tree
`int maxDepth = 10`	Default max depth of the tree
`double delta = 0.000005`	A very small delta and its default value, used for comparing against the information gain to determine if too little has been learned from this level and we should not branch any further

DTree Class Function Signatures	Description

`Decision getMaxInformationGain(`
`  vector<string>& attributes,`
`  vector< vector<double> >& data,`
`  vector<int>& outcomes,`
`  vector<int>& instances)`
	This function iterates through the *attributes* vector and calls the provided *getInformationGain*() for each attribute. If you are passing the attribute in attributes.at(i), you will pass its corresponding data from data.at(i) to the *getInformationGain*() function, along with the set of outcomes and instances being considered.
	
	This function must utilize your max-heap to save the Decisions and return the one with the max information gain at the end!
	
	Notice that the Decision class overloads several comparison operators so it works with the heap.

`void train(`
`  vector<string>& attributes,`
`  vector< vector<double> >& data,`
`  vector<int>& outcomes,`
`  vector<int>& instances)`
	This function takes the set of attributes, all data, and the list of outcomes and instances so that it can build the decision tree. It should call getMaxInformationGain to get the Decision that should be turned into the next node.

	You may find it easiest to create a recursive helper: first populate the root from the decision with the max gain and then call a recursive helper to populate the left and right children. A Decision keeps track of the instances that should be considered at each level. **Left children use the "instancesBelow" list for instances, and right children use the "instancesAbove" list**. Remember to track depth and parent when creating a new node (you can add these as parameters to your recursive helper!) You stop adding new children in a branch when you either reach maxDepth or the information gain of the max decision is below the delta.

`int classify(`
`  vector<string>& attributes,`
`  vector<double>& data)`
	Given a single instance, which is a list of attributes and their corresponding data points in parallel vectors, your goal is to return the decision tree's guess for the likely class or outcome. The return is integer since we are using binary class labels of 0 and 1.
	
	The classify operation is very similar to search for a binary search tree.
	- You start at the root and check the Decision's attribute label. You scan the *attributes* vector until you find the correct label.
	- Next, you compare the data for that attribute (at the same index in the parallel *data* vector) against the Decision point's threshold.
	- If the recorded data is less than or equal to the threshold, you go down the left subtree. Else, you go down the right subtree.
	- This operation continues until you reach the bottom of the branch (reach nullptr)
	
	Classify can be done recursively but is also quite easy to do iteratively since it is like a linked list traversal.

`string levelOrderTraversal()`	Finally, in order to understand the structure the tree has built, we can use a level-order traversal. In class, we went through an example of this algorithm with detailed pseudocode. You can implement it exactly from that example with one modification to how we print the data. Add the following line in your function when the current node is dequeued so that you can print it with an indentation level corresponding to its depth:

	`ss << string(u->depth,'\t') << u->key;` This uses a string constructor to copy the tab character a number of times equal to the depth of the current node (assuming "u" is the DNode pointer and you have saved the depth of the node). **Note: The simplest way to support this function will be to copy your working Queue implementation from PA 1 into the PA 3 folder**. You'll need the DoublyLinkedList and its Node as well. In order to save a DNode* into the Queue, you'll want to add an instantiation for DNode* to the bottom of the DoublyLinkedList class, which will require adding `#include "DecisionLogic.h"` to the top of `DoublyLinkedList.cpp`. This showcases the value of templating and how easily you can reuse templated code.
`~DTree() {`	Destructor should clear all memory being used by the DTree

Point Distribution

This part comprises 42% of the overall assignment score. That is distributed as follows:

- 2% - Bibliography
  - Update the bibliography.txt file with your references as you go to receive these points
- 5% - Manual Code Review
  - Even though only 5% is set aside for manual review of this portion, note that autograded points can be revoked if attempts to circumvent the spirit of the assignment's autograding are discovered.
- 4% - DNode
- 8% - GetMaxInformationGain
- 12% - Train and classify
- 5% - Level order traversal
- 6% - Memory leak checks

Interesting Links

In addition to using our heap to support the decision tree, heaps can be useful for other machine learning algorithms as well. For instance, a popular unsupervised learning model, k-Nearest Neighbors (kNN), works by grouping data points into clusters based on their proximity. Sorting algorithms are useful for this operation, and quicksort is among the fastest and most favored methods. However, since portions of heapsort can be performed in parallel across multiple processors, heapsort has been shown to be a faster method for certain kNN implementations.

For further reading about k-Nearest Neighbors, look at this Towards Data Science article (which is in fact the original source of our diabetes dataset for this problem). Also look at this research paper on fast heapsort-based kNNs. You will see many of the concepts we have been discussing in this class, such as sorting algorithms, Big-O complexity, and more, and you can see an example of how they are used for practical purposes.