Assignment 3

COMP 250 Winter 2021

posted: Friday, Mar. 26, 2021
due: Tuesday, April. 13, 2021 at 23:59

General Instructions

- **Submission instructions**
  - Late assignments will be accepted up to 2 days late and will be penalized by 10 points per day. Note that submitting one minute late is the same as submitting 23 hours late. We will deduct 10 points for any student who has to resubmit after the due date (i.e. late) irrespective of the reason, be it wrong file submitted, wrong file format was submitted or any other reason. This policy will hold regardless of whether or not the student can provide proof that the assignment was indeed "done" on time.
  - Don't worry if you realize that you made a mistake after you submitted : you can submit multiple times but only the latest submission will be kept. We encourage you to submit a first version a few days before the deadline (computer crashes do happen and codePost may be overloaded during rush hours).
  - These are the files you should be submitting on codePost:

    * `DecisionTree.java`

    Do not submit any other files, especially .class files. Any deviation from these requirements may lead to lost marks
- **To run the code in Eclipse, you will need to use the default package and not have a package statement.** (The reason for this constraint has to do with the use of serialization. We have tried to make it run in other packages but we not able to at the time of the assignment release. We will update you if we find a solution.)
- **Do not change any of the starter code that is given to you. Add code only where instructed, namely in the "ADD CODE HERE" block.** You should not need to add helper methods. If you wish to add helper (i.e. private!) methods, you may but only in the `DecisionTree` class.
- The assignment shall be graded automatically. Requests to evaluate the assignment manually shall not be entertained, so please make sure that you follow the instruction closely or your code may fail to pass the automatic tests. Note that for this assignment, you are NOT allowed to import any other class. **Any failure to comply with these rules will give you an automatic 0.**
- Whenever you submit your files to codepost, you will see the results of some exposed tests. These tests are a mini version of the tests we will be using to grade your work. If your code fails those tests, it means that there is a mistake somewhere. Even if your code passes those tests, it may still

contain some errors. We will test your code on a much more challenging set of examples. We highly encourage you to test your code thoroughly before submitting your final version.

- In a week we will share with you a `Minitester` class that you can run to test if your methods are correct. This class is equivalent to the exposed tests on codePost. Please note that these tests are only a subset of what we will be running on your submissions. We encourage you modify and expand these classes. You are welcome to share your tester code with other students on Piazza. Try to identify tricky cases. Do **not** hand in your tester code.
- Your code will be tested on valid inputs only.
- You will automatically get 0 if your code does not compile.
- Failure to comply with any of these rules will be penalized. If anything is unclear, it is up to you to clarify it by asking either directly a TA during office hours, or on the discussion board on Piazza.

Learning Objectives

In this assignment you will be learning about decision trees and how to use them to solve classification problems. Working on this problem will allow you to better understand how to manipulate trees and how to use recursion to exploit their recursive structure.

1 Introduction

Congratulations! You have just landed an internship at a startup software company. This company is trying to use AI techniques – in particular decision trees – to analyze spatial data. Your first task in this internship is to write a basic *decision tree* class in Java. This will demonstrate to your new employer that you understand what decision trees are and how they work.

From a quick web search, you learn that decision trees are a classical AI technique for classifying objects by their properties. One typically refers to object *attributes* rather than object properties, and one typically refers object *labels* to say how an object is classified.

As a concrete example, consider a computer vision system that analyzes surveillance video in a large store and it classifies people seen in the video as being either employees or customers. An example attribute could be the location of the person in the store. Employees tend to spend their time in different places than customers. For example, only employees are supposed to be behind the cash register.

For classification problems in general, one defines object attributes with **x** variables and one defines the object label as a **y** variable. In the example that you will work with in this assignment, the attributes will be the spatial position (x_1, x_2), and the label y will be a color (red or green).

Let's get back to decision trees themselves. Decision trees are rooted trees. So they have internal nodes and external nodes (leaves). To classify a data item (datum) using a decision tree, one starts at the root and follows a path to a leaf. Each internal node contains an *attribute test*. This test amounts to a question about the value of the attribute – for example, the location of a customer in a store. Each child of an internal node in a decision tree represents an outcome of the attribute test. For simplicity, you will only have to deal with *binary decision trees*, so the answers to attribute test questions will be either true or false. A test might be $x_1 < 5$. The answer determines which child node to follow on the path to a leaf.

The labelling of the object occurs when the path reaches a leaf node. Each *leaf node* contains a *label* that is assigned to any test data object that arrives at that leaf node after traversing the tree from the root. The label might be red or green, which could be coded using an enum type, or simply 0 or 1. Note that, for any test data object, the label given is the label of the leaf node reached by that object, which depends on the outcomes of the attribute tests at the internal nodes.

The reason that this document is larger than usual is that decision trees were not covered the pre-recorded lectures. This document should give you enough information about decision trees for you to do the assignment. If you wish to learn more about decision treees, then there are ample resources available on the web. Steer towards resources that are about decision trees in computer science, in particular, in machine learning or data mining. For example:

https://en.wikipedia.org/wiki/Decision_tree_learning

Be aware that these resources will contain more information than you need to do this assignment, and so you would need to sift through it and figure out what is important and what can be ignored. Feel free to use Piazza to share links to good resources and to resolve questions you might have. *The task of understanding what decision trees are is part of the assignment. The amount of coding you need to do is relatively small, once you figure out what needs to be done.*

1.1 Creating decision trees

To classify objects using a decision tree, we first need to have a decision tree! Where do decision trees come from? In machine learning, one creates decision trees from a labelled data set. Each data item (datum) in the given labelled data set has well defined attributes **x** and label **y**. We refer to the data set that is used to create a decision tree as the *training* set. The basic algorithm for creating a decision tree using a training set is as follows. This is the algorithm that you will need to implement for **fillDTNode()** later.

Data: data set (training)
Result: the root node of a decision tree
MAKE_DECISION_TREE_NODE(data)
if *the labelled data set has at least k data items (see below)* **then**
    if *all the data items have the same label* **then**
        create a leaf node with that class label and return it;
    else
        create a "best" attribute test question; (see details later)
        create a new node and store the attribute test in that node, namely attribute and threshold;
        split the set of data items into two subsets, data1 and data2, according to the answers to the test question;
        newNode.child1 = MAKE_DECISION_TREE_NODE(data1)
        newNode.child2 = MAKE_DECISION_TREE_NODE(data2)
        return newNode
    end
end

In the program, k is an argument of the decision tree construction minSizeDatalist.

1.2 Classification using decision trees

Once you have a decision tree, you can use it to classify new objects. This is called the *testing* phase. For the testing phase, one can use data items from the original data used for training (above) or one can use new data. Typically when a decision tree is used in practice, the test objects are *unlabelled*. In the surveillance example earlier, the system would test a new video and try to classify people as employees versus customers. Here the idea is that one does not know the correct class for each person. Let's consider this general scenario now, that we are given a decision tree and the attributes of some new unlabelled test object. We will to use the decision tree to choose a label for the object. This is done by traversing the decision tree from the root to a leaf, as follows:

Data: A decision tree, and an unlabelled data item (datum) to be classified
Result: (Predicted) classification label
CLASSIFY($node, datum$) {
if *node is a leaf* **then**
    return the label of that node i.e. classify;
else
    test the data item using question stored at that (internal) node, and determine which $child$ node to go to, based on the answer ;
    return CLASSIFY($child, datum$);
end
}

2 Instantiating the decision tree problem

For this assignment, the problem is to classify points based on their position. Each datapoint has an array of attributes **x**, and a binary label **y** (0 or 1). For this section, we will focus on datapoints with only two attributes.
A graphical representation of example of a data set looks like this. (For the graphs, the attribute value x[0], is represented as x1 and x[1] as x2.) For those who print out the document in color, the red symbols can be label 0 and the green symbols can be label 1. For those printing in black and white, the (red) disks are label 0 and the (green) $\times$'s are label 1.

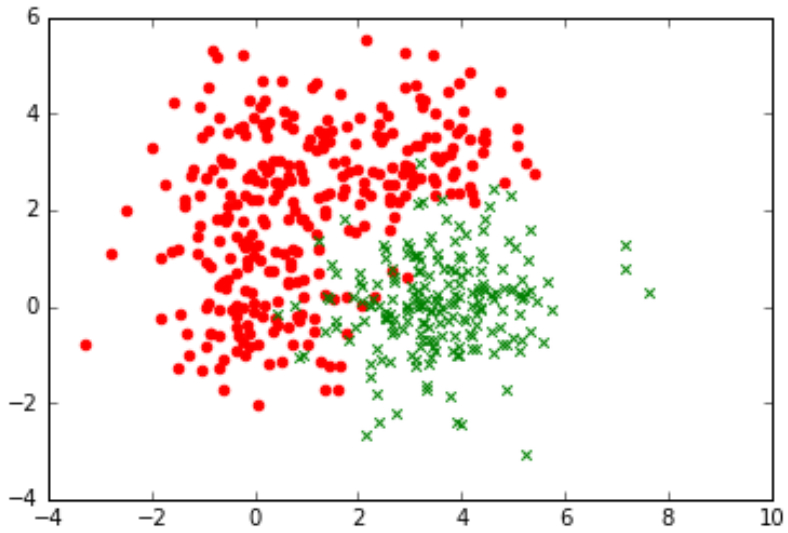


Figure 1: x_1 is horizontal and x_2 is vertical coordinate. Note that the points are intermixed. There is no way to draw a horizontal or vertical line or any curve for that matter that could split the data.

2.1 Finding a good split

Now that we have an idea of what the data are, let us return to the question of how to split the data into two sets when creating a node in a decision tree. What makes a 'good' split? Intuitively, a split is good when the labels in each set are as 'pure' as possible, that is, each subset is dominated as much as possible by a single label (and the dominant label differs between subsets). For example, suppose this is our data:

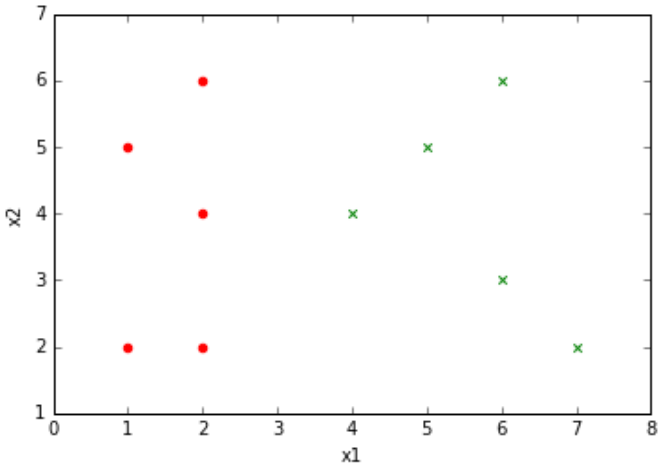
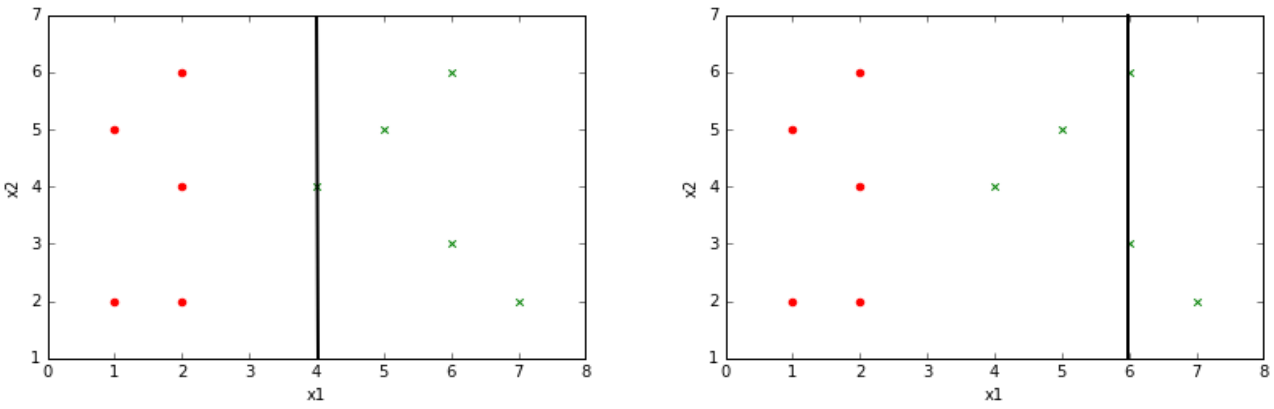


Figure 2: What would be a good split of this data?

Two of many possible splits we could make are shown in Fig. 3. Fig 3-a splits the data into two sets based on the test condition ($x_1 < 4$), i.e. true or false. (By definition, the green symbol that falls on this line is considered to be in the right half since the inequality is strict.) This is a good split in that all data points for which the test condition is false have the same label (green) and all data points for which the test condition is true have the same label (red), and the labels differ in the two subsets.

The split condition ($x_1 < 6$) in Fig. 3-b is not as good, since the subset for which the condition is true contains datapoints of both labels.

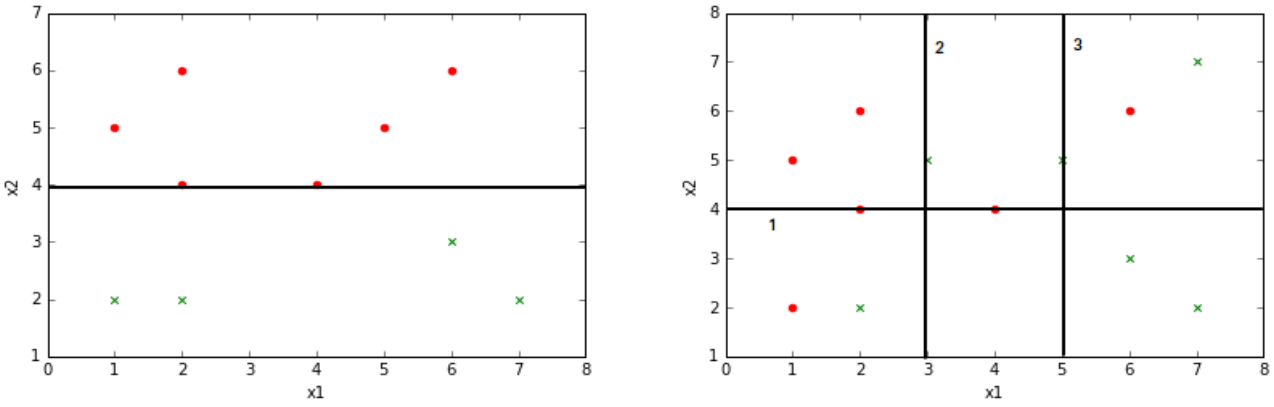


(a) (b)

Figure 3: Example of different splits on the x_1 attribute.

Splits can be done on either of the attributes. For the example in Fig. 4-a, a good split would be defined by the test condition $(x_2 < 4)$.

The situation is more complicated when the data points cannot be separated by a threshold on x_1 or x_2, as in Fig 4-b, however. It is unclear how to decide which of the three splits is best. We need a quantitative way of deciding how to do this.



(a) (b)

Figure 4: (a) Example of a data set and a split using the x_2 attribute, where the two subsets have distinct labels. (b) An example of a data set in which there is no way to split using either an x_1 or x_2 value, such that the two subsets have distinct labels.

2.2 Entropy

To handle more complicated situation, ones needs a quantitative measure of the 'goodness' of a split, in terms of the impurity of the labels in a set of data points. There are many ways to do so. One of the most common is called *entropy*.[1] The standard definition[2] of entropy is:

$$H = -\sum_i p_i \log_2(p_i) \tag{1}$$

where $p(i)$ is a function such that $0 \leq p_i \leq 1$ and $\sum_i p_i = 1$. Note that the minus sign is needed to make H positive, since $\log_2 p_i < 0$ when $0 < p_i < 1$. Also, note that if $p_i = 0$ then $p_i \log_2(p_i) = 0$ since that is the limit of this expression as $p_i \to 0$. (Recall l'Hopital's Rule in Calculus 1.)

For the special case that there are two values only, namely p_1 and $p_2 = 1 - p_1$, entropy H is between 0 and 1, and we can write H as a function of the value $p = p_1$. For a plot of this function $H(p)$, see:

https://en.wikipedia.org/wiki/Binary_entropy_function

[1] Entropy is an extremely important concept in science. It has its roots in thermodynamics in the 19th century. In the 20th century, "information entropy" was one of the basic for techniques in electronic communication (telephones, cell phones, internet, etc). In computer science, information entropy is heavily used in data compression, cryptography, and AI.

[2] Such functions p_i are often used to model probabilities, as you will learn if you take MATH 323 or MATH 203 for example.

In this assignment, we use entropy to choose the best split for a data set, based on its labels. We borrow the formula for entropy and apply it to our problem as follows:

$$H(D) = -\sum_{y \in L} p(y) \log_2 p(y) \tag{2}$$

where

- L is the set of labels, and y is a particular label
- D is a data set; each data point has two attributes and a label i.e. (x_1, x_2, y)
- $H(D)$ is the entropy of the dataset D
- $p(y)$ is the fraction of data points with label y from dataset D.

Since L consists of only two labels, entropy is between 0 and 1. Entropy is 0 if $p(y)$ takes values 0 and 1 for the two labels. Entropy is 1 if $p(y) = 0.5$ for both labels. Otherwise it has a value strictly between 0 and 1. See plot in the link above.

2.3 Using entropy to define a good split

During the training phase, when one constructs the decision tree, a node is given a data set D as input. If D has entropy greater than 0, then we would like to split the data set into two subsets D_1 and D_2. We would like the entropy of the subsets to be lower than the entropy of D. The subsets may have different entropy, however, so we consider the average entropy of the subsets. Moreover, because one subset might be larger than the other, we would like to give more weight to the larger subset. So we define the *average entropy* like this:

$$H(D_1, D_2) \equiv w_1 \times H(D_1) + w_2 \times H(D_2)$$

where w_i is the fraction of the points in subset i,

$$w_i = \frac{number\ of\ datapoints\ in\ D_i}{number\ datapoints\ in\ D,\ namely\ D_1 + D_2}$$

and i is either 1 or 2. Note that $w_1 + w_2 = 1$.

NOTE: DO NOT use the formula : w_2 = 1 - w_1, although correct, this leads to a numerical approximation error. For each of the weights (w_1, w_2) use the formula mentioned above separately.

ASIDE: (We mention the following because you will likely encounter it in your reading.) When building a decision tree, one often considers the difference $H(D) - H(D_1, D_2)$, which is called the *information gain*. For example, one may decide whether or not to split a node based on whether the information gain is sufficiently large. In this assignment, you will instead use a different criterion to decide whether to split a node when building the decision tree Your criterion will be based on the number of data items in D, as will be discussed later.

2.4 Example

Let us now return to the examples we saw earlier and use entropy to discuss which split is best. Recall the example of Fig. 4-b. To calculate the entropy of the dataset D before split, note there are two classes with 6 points in each of the classes. So the fraction of points in each class is 0.5 and the entropy is:

$$H(D) = -\frac{1}{2}\log_2\left(\frac{1}{2}\right) - \frac{1}{2}\log_2\left(\frac{1}{2}\right) = 1 \tag{3}$$

- Split 1 breaks the dataset into two sub datasets, where the subdataset on top contains 8 points (5 red dots , 3 green crosses) and the one below has 4 points (1 red dot, 3 green crosses). Calculating average entropy $H(D_1, D_2)$ after the split yields:

$$\frac{4}{12}\left(-\left(\frac{1}{4}\log_2\left(\frac{1}{4}\right) + \frac{3}{4}\log_2\left(\frac{3}{4}\right)\right)\right) + \frac{8}{12}\left(-\left(\frac{5}{8}\log_2\left(\frac{5}{8}\right) + \frac{3}{8}\log_2\left(\frac{3}{8}\right)\right)\right) = 0.906 \tag{4}$$

- Split 2 breaks the dataset into two sub datasets, where the sub dataset on the left has 5 datapoints (4 red dots, 1 green cross) and the other one has 7 datapoints (2 red dots , 5 green crosses). The average entropy $H(D_1, D_2)$ after the split is:

$$\frac{5}{12}\left(-\left(\frac{1}{5}\log_2\left(\frac{1}{5}\right) + \frac{4}{5}\log_2\left(\frac{4}{5}\right)\right)\right) + \frac{7}{12}\left(-\left(\frac{2}{7}\log_2\left(\frac{2}{7}\right) + \frac{5}{7}\log_2\left(\frac{5}{7}\right)\right)\right) = 0.803 \tag{5}$$

- Split 3 breaks the dataset into 2 sub datasets, where the sub dataset on the left has 7 datapoints (5 red dots, 2 green crosses) and the other one has 5 datapoints (1 red dot , 4 green crosses). The average entropy $H(D_1, D_2)$ after the split is:

$$\frac{7}{12}\left(-\left(\frac{2}{7}\log_2\left(\frac{2}{7}\right) + \frac{5}{7}\log_2\left(\frac{5}{7}\right)\right)\right) + \frac{5}{12}\left(-\left(\frac{1}{5}\log_2\left(\frac{1}{5}\right) + \frac{4}{5}\log_2\left(\frac{4}{5}\right)\right)\right) = 0.803 \tag{6}$$

So, split 2 and 3 have lower average entropy than split 1. It is no problem that split 2 and 3 have the same average entropy. Either one can be chosen as the 'best split'. For the assignment, select the first encountered best split (when the check for the best split starts from the first attribute (`x[0]`) and proceeds from there and for a given attribute the check starts from the first datapoint and proceeds thereafter.

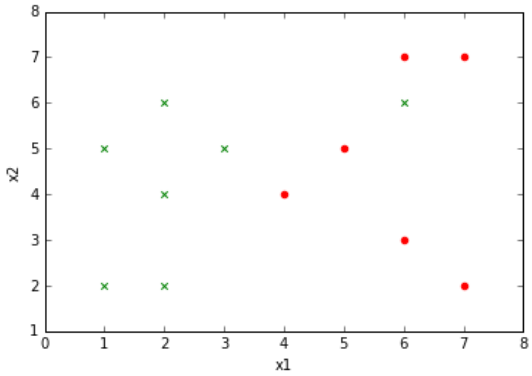
2.5 Finding the best split

We can use entropy to compare the 'goodness' of different splits. The simplest way to define *the best split* is just to consider all possible splits and choose the one with the lowest average entropy. In this assignment, you will take this brute force approach. You will consider each attribute x_1 and x_2 and each of the values of that attribute for points in the data set. You will compute the average entropy for splitting on that attribute value, and you will choose the split that gives the minumum.

Data: A Dataset
Result: An attribute and a threshold pair
FIND_BEST_SPLIT($data$) {
best_avg_entropy := inf;
best_attr := -1;
best_threshold := -1;
for *each attribute in* x **do**
    for *each data point in list* **do**
        compute split and current_avg_entropy based that split;
        if *best_avg_entropy* $>$ *current_avg_entropy* **then**
            best_avg_entropy := current_avg_entropy;
            best_attr := attribute;
            best_threshold := value;
        end
    end
end
return (best_attr, best_threshold)
}

Note the order of the two **for** loops! If you use the opposite order, you might get a different tree (wrong answer).

So, far we have seen how to create the decision tree and what are the intuitions behind the math needed to get the split to build the tree. Let us consider another example. Consider the data shown below.



A decision tree for this dataset should have splits as shown below.

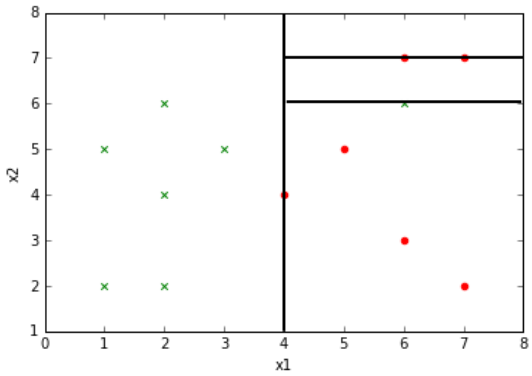


Figure 5: Example of an overfit decision tree on the outlier dataset

But does this seem right? Given that all the points around it are of class 1 and the other points of the same class are far to the right, it might be possible that the datapoint labeled class 2 at the left side of the graph (6,6) is an outlier or an anomalous reading in the data. Points like these are most of the time, if not always present in a dataset and care should be taken so that our decision tree does not try too hard to reduce the impurity of the sub datasets and in the process actively keep splitting the data so as to try to classify the outliers into purer groups.

This phenomenon described above is called overfitting. The algorithm – in this case the construction of a decision tree – tries to find a "model" that accounts for all of the data, even the data which might be garbage for some reason (noise, or due to some error in the program or device that produced the data).

2.6 Preventing Overfitting

There are different ways to prevent overfitting. The method that you will use is called *early stopping,* namely stop splitting nodes in the decision tree when the number of data points in a subset is smaller than some predetermined number. The issue of overfitting is very important in decision trees and in machine learning in general, but the details are beyond the scope of this assignment.

3 Instructions and starter code

The starter code contains three classes:

Datum.java

This class holds the information of a single datapoint. It has two variables, x and y. x is an array containing the attributes and y contains the label.

The class also comes with a method `toString()` which returns a string representation of the attributes and label of a single datapoint.

DataReader.java

This class deals with three things. The method `read_data()` reads a dataset from a csv[3], and splits the read dataset into the training and test set, using `splitTrainTestData()` file. It also has methods that deal with reading and writing of "serialized" decision tree objects.

DecisionTree.java

This is the main class which deals with the creation of a decision tree and classification of datapoints using the created decision tree. You will be implementing some of the methods in this class.
Let us go through the different members and attributes of the class:

- The constructor builds a decision tree by calling the `fillDTNode()` method on a dataset. It is given a list of data points and a parameter that specifies the minimum number of datapoints that has to be present in a given dataset to qualify for a split. This minimum number is used to reduced the chances of overfitting, as discussed above.
- There is a root node field, called `rootDTNode`, by which other nodes can be accessed.
- There is a field called `minSizeDatalist` used to store the minimum number of datapoints that should be present in the dataset so as to initiate a split.
- There is a subclass class `DTNode`. This class is used to represent a single node of a decision tree. There are two types of nodes: the internal nodes which define an attribute and a threshold which help in classification, and leaf nodes which determine the labels of those data points whose attributes obey the threshold conditions of the ancestor internal nodes leading up to the leaf node.

  The DTNode contains the following members:

  - `leaf`: a boolean variable that indicates whether this node object is a leaf or not.
  - `label`: an integer variable that indicates the label of the node. The label of the node indicates class of a datapoint that reaches that particular leaf node after traversing the tree. This is valid only if the node is a leaf node. The classes for this assignment are simply 0 or 1.
  - `attribute`: The attribute, $(x_1, \ldots, x_n)$ on which the dataset is split in that particular node. The value of the attribute is one of $\{1, \ldots, n\}$. The attribute value is meaningful only for an internal node.

    In the code, the attributes are `x[0], x[1], .., x[n-1]` and their attribute values are $0, 1, \ldots, n-1$, respectively..
  - `threshold`: This holds the value of the attribute at which the split is done. This is also only meaningful for an internal node.
  - `left, right` These two variables are of type DTNode, and they represent the two children of an internal node. At the classify stage, the left child leads to a decision tree node that handles

[3] According to https://en.wikipedia.org/wiki/Comma-separated_values, comma-separated values (CSV) file is a delimited text file that uses a comma to separate values. A CSV file stores tabular data (numbers and text) in plain text. Each line of the file is a data record. Each record consists of one or more fields, separated by commas.

the case that the value of the attribute is less than the threshold, and right handles the case that the value is greater than or equal to the threshold. For a leaf node, they are both null.

The class `DTNode` also contains a few member methods that helps in building the tree.

- `fillDTNode()`: This is the method that does all the heavy lifting in the entire assignment. Given a list of data points and a minimum size for splitting (see earlier), this recursive function creates the entire decision tree. A detailed description is present in the comments of the code.
- `findMajority()`: Given a list of datapoints D, this method returns a label for that set. This method is called during training (construction of the decision tree). It can happen that the size of a dataset falls below the minimum size mentioned in `minSizeDatalist`, but still contains datapoints from more than one class. In such a case, a leaf node is created. To determine the label for this leaf node, the method goes through all the points and finds the majority label, i.e. the most common label for those points. This label will be used later at the classification phase, when a new data point reaches that leaf node. When choosing the label for a leaf node, if there is no majority (a tie), then label with the smallest value is returned.
- `classifyAtNode()`: At the testing phase, given a datapoint with only its attributes (no label), this method uses the label specified by the decision tree leaf node (as was determined at training time by the `findMajority()` method.
- `equals()` : Given another node, this method checks if the tree rooted at the given node is equal to the tree rooted at the calling node. The definition of 'equality' is elaborated in the next section.

- `calcEntropy()`: Given a dataset, this function calculates the entropy of the dataset.
- `classify()`: Given a datapoint (without the label), predicts the label of the datapoint. The only difference between this method and `classifyAtNode()` is that `classifyAtNode()` does the classification on its member DTNode, whereas for `classify()` the DTNode is the root of the created decision tree.
- `checkPerformance()`: Given a dataset where the datapoints have both attributes and labels, this method runs the `classify()` function on all of the datapoints (using only the attributes) and compares the label that is given by the decision tree with the "ground truth" label for that data point. The method returns the fraction of datapoints that were predicted wrong, in the form of a string.
- `equals()`: Given two decision trees, this method checks if the two trees are equal or not. It returns a boolean value.

Your task

You need to implement three methods from the DTNode class. None of the methods depend on each other. We suggest that you implement `equals()` first.

1. `DTNode.equals()` **(30 points)**
   This method compares two DTNodes. Given another DTNode object, it checks if the tree that is rooted at the calling DTNode is equal to the tree rooted at DTNode object that is passed as the

parameter.
Two DTNodes are considered equal if:

(a) a traversal (e.g. preorder) of each of the two trees encounters nodes that are equal;

(b) internal node : the thresholds and attributes should be same;

(c) leaf node : the labels should be same.

Note that the tester you are given uses this equals method to check if the tree you implement in the second part matches with the actual solution.

2. `DTNode.fillDTNode()` **(50 points)**
   This method takes in a datalist (i.e. an arraylist of objects of type Datum) and returns the calling DTNode object as the root of a decision tree trained using the datapoints present in the input datalist.

3. `DTNode.classifyAtNode()` **(20 points)**
   This method takes in a datapoint (excluding the label) in the form of an array of type double (like Datum.x) and should return its corresponding label (int).

4 Data

The different datasets used in the assignment are shown below. If you look at the `DataReader()` class, you will note that only half of the data in each plot is used in the training. The other half is used to test the performance of the decision tree. This testing phase is not part of the assignment, but we give you some performance data so that you can appreciate the differences in the data sets.

For each of the three data sets, we list the performance of the decision tree in classifying the other half (test set) of the data. We show both the fraction of training points that are misclassified (error) and the fraction of the test points that are misclassified, and we do so for different values of the variable `minSizeDatalist`.

4.1 Highly overlapping data :

Notice that the training error is 0 when the minimum size is 1 and rises as `minSizeDatalist` increases. However, the test error is near .5 (50 percent) for all values of `minSizeDatalist`.

minSizeDatalist : 1	Training error : 0.000	Test error : 0.495
minSizeDatalist : 2	Training error : 0.000	Test error : 0.495
minSizeDatalist : 4	Training error : 0.030	Test error : 0.495
minSizeDatalist : 8	Training error : 0.105	Test error : 0.520
minSizeDatalist : 16	Training error : 0.200	Test error : 0.515
minSizeDatalist : 32	Training error : 0.235	Test error : 0.515
minSizeDatalist : 64	Training error : 0.310	Test error : 0.525
minSizeDatalist : 128	Training error : 0.390	Test error : 0.490

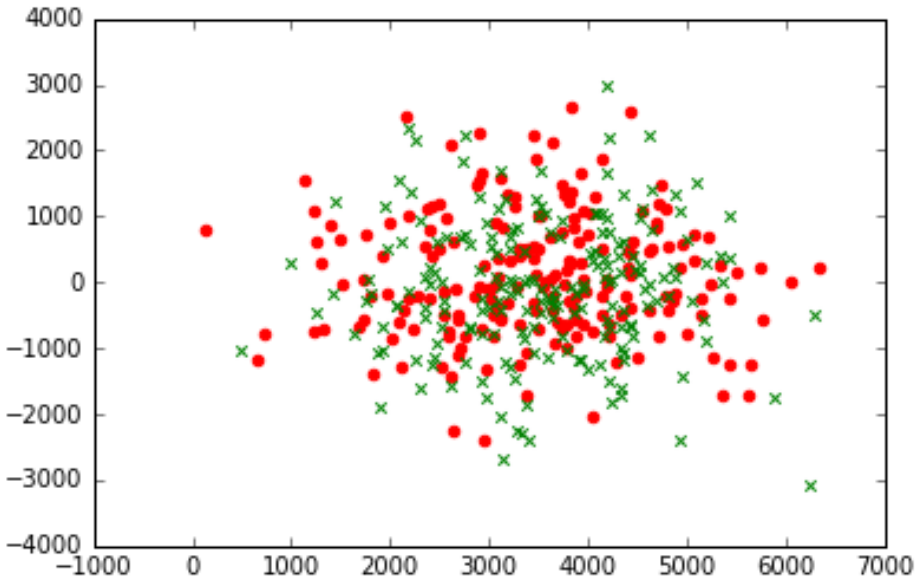


Figure 6: Plot of data present in data_high_overlap.csv

4.2 Partially overlapping data :

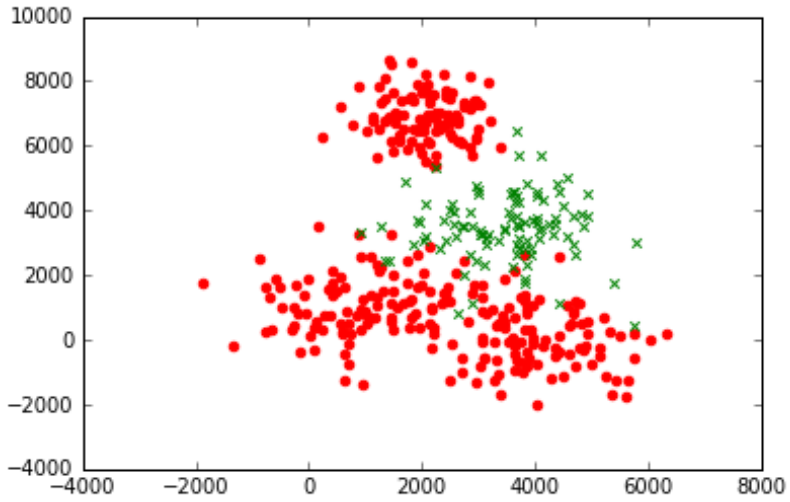


Figure 7: Plot of data present in data_partial_overlap.csv

In this case, 10 or 12 of the 200 test data points are misclassified for most values of `minSizeDatalist`. The error rises considerably when `minSizeDatalist` is 128.

minSizeDatalist : 1 Training error : 0.000 Test error : 0.050
minSizeDatalist : 2 Training error : 0.000 Test error : 0.050
minSizeDatalist : 4 Training error : 0.015 Test error : 0.050
minSizeDatalist : 8 Training error : 0.035 Test error : 0.060
minSizeDatalist : 16 Training error : 0.045 Test error : 0.050

minSizeDatalist : 32 Training error : 0.045 Test error : 0.050
minSizeDatalist : 64 Training error : 0.075 Test error : 0.050
minSizeDatalist : 128 Training error : 0.255 Test error : 0.245

4.3 Minimal overlapping data :

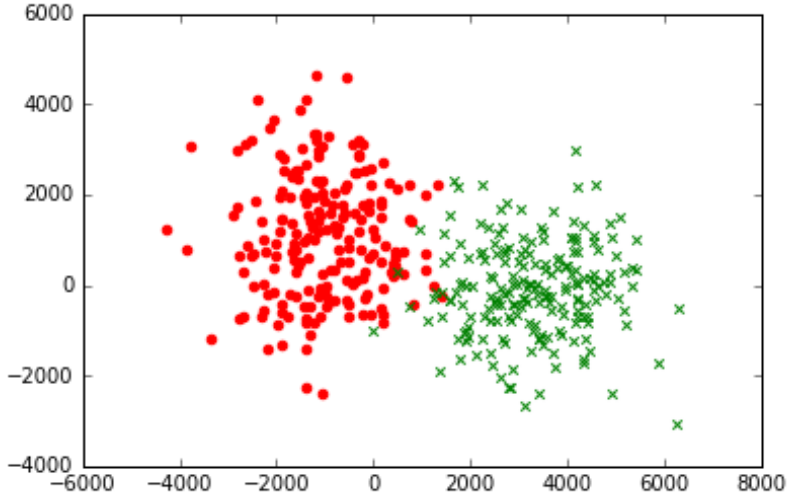


Figure 8: The data points here have very little overlap. We are loosely calling the overlap "minimal", although truly minimal would mean zero overlap. We leave some overlap so that overfitting can potentially occur.

Note exactly one (of 200) training data points is misclassfied when `minSizeDatalist` is 4 or more. The test error is roughly constant. It is slightly larger for small values of `minSizeDatalist`.

minSizeDatalist : 1 Training error : 0.000 Test error : 0.040
minSizeDatalist : 2 Training error : 0.000 Test error : 0.040
minSizeDatalist : 4 Training error : 0.005 Test error : 0.040
minSizeDatalist : 8 Training error : 0.005 Test error : 0.035
minSizeDatalist : 16 Training error : 0.005 Test error : 0.035
minSizeDatalist : 32 Training error : 0.005 Test error : 0.035
minSizeDatalist : 64 Training error : 0.005 Test error : 0.035
minSizeDatalist : 128 Training error : 0.005 Test error : 0.035