Project 3
Decision Tree

Out: Thursday, April 4
In: Wednesday, April 17, 11:59 PM
"If a decision tree falls in a forest, and no one is there to hear it, is the classification correct?"
-Emily Magavern

1 Installing, Handing In, Demos

1. To install, type `cs0160_install decisiontree` into a shell. The script will create the appropriate directories and deposit stencil files into them.
2. To compile your code, type `make` in your project directory. To run your code and launch the visualizer, type `make run` in the same directory. Make sure that your `Makefile` is in the same directory as your code.
3. To run tests, run `make run_tests` from your project directory.
4. To hand in your project, go to the directory you wish to hand in, and type `cs0160_handin decisiontree` into a shell.
5. To run a demo of this project, type `cs0160_runDemo decisiontree` into a shell. Training and testing data to use with the demo can be found in `/course/cs0160/lib/decisiontree-data`.
6. Documentation for support code used in this project can be found here: `http://cs.brown.edu/courses/csci0160/static/files/docs/doc/decisiontree/index.html`. This is also linked off of the class website.
7. Remember to not include any identifying information in your hand in.

2 Using Eclipse

As with previous projects, we recommend that you use Eclipse (or some other IDE) to complete this project. However, please make sure that you are using Eclipse Luna, not Eclipse Photon. You can launch Eclipse Luna by running `eclipse &`. In order to set up your project and make Eclipse work with the support code, you'll need to do the following:

- Select `File` → `New` → `Java Project`
  - Enter "decisiontree", lowercased, for the project name.

  - Un-check the box saying "Use default location." By the "Location" box, either click "Browse" or enter the full filepath to your project folder. On a department machine, this path is `/gpfs/main/home/<your-login>/course/cs0160/<project_name>`. If you are working locally, the path will be different.
  - Click "next"
  - Under the "libraries" tab choose "Add External JARs..."
  - If you are working on a department machine, navigate to `/course/cs0160/lib`. If you are working locally on your own machine, ensure that you have copied the .jar files we've used for previous projects over to your local machine, and then navigate to where they are stored.
  - Select `cs0160.jar, junit-4.12.jar, hamcrest-core-1.3.jar`. Note that you do not need to include NDS4 for this project.
  - Click "Finish"
  - If it isn't already made for you, use `File` → `New` → `Source Folder` to create a new source folder in your new project named "src"
  - Use `File` → `New` → `Package` to create a new package in your new source folder named "decisiontree", and move all the stencil Java files into this package. Ignore any errors that may arise when moving the files, but if you have compilation errors that arise after the move, you'll need to fix those before your code will run.

- Right-click on App.java and select `Run As` → `Java Application`. Now you can run your program by pressing the green play button at the top of your screen and selecting "Java application" if prompted
- To run the tests in eclipse, you can right-click on `TestRunner.java` and click `Run As` → `Java Application`.
- To configure your Eclipse projects to run over FastX or SSH, follow these setup steps
  - Right click on the package icon next to the project name. Go to properities.
  - Go to Run/Debug Settings, select the main window App and click Edit.
  - Go to the arguments tab and, and enter `-Dprism.order=sw` in the VM arguments block
  - Hit Apply and OK. You should be all set to work on Decision Tree!

3 Introduction

Please **read the entire handout carefully** before you begin - it is full of important information!

3.1 Silly Premise

Its a Friday night and your parents drive you up to the nearest BlockBuster. You go through the door and immediately see all of your favorite movies. Problem is, you cant break it down to the one movie that your parents will rent out for you. You need to figure out a way to decide which movie you want taking into account all of their attributes. You also get to pick one movie snack! Tell us in your README what that would be! :)

3.2 Your Task

In this project, you will implement the ID3 algorithm and use it to generate a decision tree from a dataset. We've provided a stencil for the `MyID3.java` class. You'll need to do the following:

- Fill in the `id3Trigger` method in `MyID3.java`. This method is called when you click 'Train' in the visualizer. It takes in the data you'll be using to train your tree and it should build your tree and return the root. See the Javadocs (link) to understand the data object you are given as a parameter. You are **required** to implement the algorithm **recursively**.
- Factor out your code into **helper methods** that you can test - we will be grading your algorithm design, and we expect frequent use of helper methods. Do your best to avoid repeated code.
- Test your program! You will do this by using the visualizer to see how your tree performs with testing data. You will also create your own testing suite by writing JUnit tests. See section 7 for more details.

3.3 README

You're required to hand in a README text file (which **must be named README.txt**) that documents any notable design choices or known bugs in your program. Remember that clear, detailed, and concise READMEs make your TAs happier when it counts (right before grading your project). Your README should be saved in the same directory as your Java files. Please refer to the README Guide in the Docs section of the CS16 Website (link).

4 Decision Trees

Decision trees are one of the most common and successful kinds of ML models. A decision tree represents a function that takes as input a set of attribute values describing something and returns a classification.

But before we can use a decision tree to classify things, we must first build it! And we do this by "learning" the decision tree using **training data**. Training data is a collection

of examples for which the classification is already known. Once a decision tree is "learned", it can be used to compute the classification of new examples that it has not seen before (i.e., that were not part of the training data). In this project, we will work with examples that can only have one of two classifications.

5 Vocabulary

These are terms you'll see throughout this handout. Here are some basic definitions; feel free to refer back to this section if needed!

- Attribute: A quality that is recorded about every example and is used to predict an example's classification.
- Example: A particular "datum" from the data set. That is, a set of values for each of the data set's attributes, and a positive or negative classification. In the scenario from lecture, an example represents one individual's choice to wait for a table at the restaurant depending on the value of certain attributes (was it raining, how full the restaurant was, etc.) at that moment.
- Value: A value that an attribute can have. Every example has one value for each attribute. The values for the attribute "Price" are {"$", "$$", "$$$"}.
- Classification: The ultimate "decision" made for each example. Can be any pair of strings, such as "Yes" and "No" or "2000's" and "90's".
- Entropy: A quantification of the homogeneity of a set.
- Remainder: The amount of entropy remaining in the data after "splitting" on some attribute.
- Information Gain: The amount by which entropy is reduced upon "splitting" on some attribute.
- Importance: An attribute's importance is measured by the amount of information gained by splitting on it. The most important attribute has the highest gain.
- Training Data: A subset of a data set used to create the decision tree.
- Testing Data: A subset of the data set, distinct from the training data, that is run on the tree (created with the training data) to test it. For each example, the classification the tree predicts for it is compared to the actual classification in order to measure accuracy.
- Splitting on an attribute: partitioning a set of examples according to the values they take on the attribute. The number of subsets in the partition will equal the number of values the attribute can take.

6 Training Data

The training data is a collection of examples, where each example is described using a set of attributes and a classification. Each attribute will be one of a set of **values** and the classification is one of two possible outcomes. In our restaurant scenario from lecture, every example is described using the following attributes (with their possible values):

1. *Alternate*: {Yes, No}: whether there is a suitable alternative restaurant nearby.
2. *Bar*: {Yes, No}: whether the restaurant has a comfortable bar area to wait in.
3. *Fri/Sat*: {Yes, No}: true on Fridays and Saturdays.
4. *Hungry*: {Yes, No}: whether we are hungry.
5. *Patrons*: {None, Some, Full}: how many people are in the restaurant.
6. *Price*: {$, $$, $$$}: the restaurant's price range.
7. *Raining*: {Yes, No}: whether it is raining outside.
8. *Reservation*: {Yes, No}: whether we made a reservation.
9. *Type*: {French, Italian, Thai, burger}: the kind of restaurant.
10. *Wait Estimate*: {0-10, 10-30, 30-60, > 60} the wait in minutes estimated by the host.

In this scenario, there are two possible classifications: Yes and No. This means a person can decide to either wait for a table or not. Note that the examples in the training data can then be partitioned into two different groups according to their classification: the examples with classification Yes and the examples with classification No. We will refer to the examples with Yes classification as positive examples and to the ones with No classification as negative examples. That is, a person that decides to patron the restaurant represents a positive example whereas one that does not is a negative example.

Figure 1, seen in lecture, is an example of a training dataset.

Note on positive/negative. Not all datasets have classification {Yes, No}. For some datasets the classification may be {true, false}, $\{0, 1\}$, {high, low}, {2000's, 90's} or any other set of two values. In the formulas we describe below we use the terms positive and negative but the "meaning" of positive and negative is *completely arbitrary*. In other words, in the restaurant example, it does not matter whether positive represents Yes and negative represents No or vice versa. Similarly, in the 2000's/90's scenario, it does not matter whether positive represents 2000's and negative represents 90's or vice versa.

7 The ID3 Algorithm

You will be implementing the ID3 algorithm to create a decision tree. At a high level, the algorithm creates a tree with attribute names as internal nodes, attribute values on the tree's branches, and leaves with classifications. Run the demo to see what this looks like!

Ex.	Input Attributes										Classif.
	Alt	Bar	Fri	Hun	Pat	Price	Rain	Res	Type	Est	
1	Yes	No	No	Yes	Some	$$$	No	Yes	French	0-10	true
2	Yes	No	No	Yes	Full	$	No	No	Thai	30-60	false
3	No	Yes	No	No	Some	$	No	No	Burger	0-10	true
4	Yes	No	Yes	Yes	Full	$	Yes	No	Thai	10-30	true
5	Yes	No	Yes	No	Full	$$$	No	Yes	French	>60	false
6	No	Yes	No	Yes	Some	$$	Yes	Yes	Italian	0-10	true
7	No	Yes	No	No	None	$	Yes	No	Burger	0-10	false
8	No	No	No	Yes	Some	$$	Yes	Yes	Thai	0-10	true
9	No	Yes	Yes	No	Full	$	Yes	No	Burger	>60	false
10	Yes	Yes	Yes	Yes	Full	$$$	No	Yes	Italian	10-30	false
11	No	No	No	No	None	$	No	No	Thai	0-10	false
12	Yes	Yes	Yes	Yes	Full	$	No	No	Burger	30-60	true

Figure 1: Training data

Our goal is to create a tree that is: (1) consistent with the examples in the training data; (2) as small as possible; and (3) does well on examples it has not previously seen. The ID3 algorithm uses a greedy divide-and-conquer strategy to decide which attribute (or classification) should be assigned to a node (or leaf). At each internal node, it does this by choosing the attribute with the most "importance" which, intuitively, means that the attribute creates a "split" of the training examples that is as homogeneous as possible.

Let's go through an example using the training data above. We will refer to the set of examples in the training data as E. Now suppose the algorithm wanted to determine the importance of the attribute "Pat" with respect to the examples in E. The Pat attribute has three values: None, Some and Full. We can use these values to *split* the examples in E into three subsets: a None subset $S_N \subseteq E$ that includes the examples with Pat=None, a Some subset $S_S \subseteq E$ that includes the examples with Pat=Some and a Full subset $S_F \subseteq E$ that includes the examples with Pat=Full. Now, within each of these subsets, we can count the number of positive examples and the number of negative examples. Intuitively, we will consider an attribute "important" if it leads to subsets that are homogeneous in the sense that they are either all positive or all negative.

So let's see how the Pat attribute splits the training data. If we consider the None subset $S_N = \{7, 11\}$ (here the numbers refer to the examples), we can see that it includes all negative examples. If we look at the Some subset $S_S = \{1, 3, 6, 8\}$ we can see that it includes all positive examples. Finally, if we look at the Full subset $S_F = \{2, 4, 5, 9, 10, 12\}$ we can see that it includes two positive examples (4 and 12) and four negative examples (2, 5, 9 and 10).

What is all this telling us? We can interpret the fact that the None subset S_N is homogeneous as the training data telling us that it is a safe bet to classify examples with

Pat=None as negative. Similarly, we can interpret the fact that the Some subset S_S is homogeneous as the training data telling us that it is a safe bet to classify examples with Pat=Some as positive. But what can we say about the Full subset S_F since it is not homogeneous? Well, we can interpret its heterogeneity as the training data telling us that there is no safe bet and we should consider another attribute *on that subset* to see if we can find a good split for it. In other words, we should recur. For this reason, your implementation **MUST be recursive**. See the lecture slides for the pseudocode of ID3.

There are four cases to consider for each recursive problem:

1. The set of examples on which you recurred is empty: No example has been found with this particular combination of values (although the combination is possible). In this case, return the majority classification (more frequent classification) of the parent's set of examples. This is the reason we also pass the `parent_examples` parameter to the algorithm.

2. The set of examples on which you recurred is all positive or all negative: We're done, and we add a leaf with the corresponding classification.

3. There are no attributes left but there are still positive and negative examples: This means that there are examples with this combination of attribute values, but they resulted in different classifications. This means that there is an error or noise in the data, meaning that we can't algorithmically interpret the data based on the values. When this happens, we just return the majority classification of the examples we have.

4. The set of examples on which you recurred includes some positive and some negative examples: Choose the most important attribute (see section below), split the examples using this attribute, recur on each subset created, associate the attribute to a new node N, attach this node to the nodes and/or leaves returned by the recursive calls, and return the node N.

Note: When finding the majority classification, you can break ties however you choose. The demo picks the negative classification when there is a tie, but you are allowed to choose the positive or decide randomly. This will likely impact your accuracy on the short data set, because it is so small (see section 7), and your tree may differ from the demo on one leaf node.

Essentially, the most important attribute is chosen based on the current examples. That attribute becomes the current node. Then, each of the possible values for that attribute (for Pat, these would be None, Some, and Full) become branches from that node. Then, the algorithm recurs to create nodes for these branches, except this time only taking in the examples that had that branch's value. This recurrence happens until we reach one of the base cases that creates a leaf.

7.1 Information Gain, Remainder & Entropy

Determining which attribute is the "most important" is tricky. Formally, attribute importance is determined by how well its values split the data. An attribute with maximum importance would split the examples (as described above) into subsets that are homogeneous, i.e., such that each subset contains only positive or only negative examples. If this is the case, then we can create branches from a node with this attribute to leaves with a classification (since all the examples with that particular value have a classification). An attribute with low importance would split the examples into subsets that all had the same number of negative and positive examples. The importance of an attribute will be quantified using the notion of **information gain**. But before we can describe what the information gain is, we first have to describe the notions of **entropy** and of **remainder**.

Entropy. The homogeneity of a subset of examples can be quantified with the notion of *entropy*.[1] Entropy is measured as a value from 0 to 1. A high amount of entropy (i.e., closer to 1) will correspond to a low amount of homogeneity (or a high amount of heterogeneity). A low amount of entropy (i.e., closer to 0) will correspond to a high amount of homogeneity (or low amount of heterogeneity).

For any subset of examples $S \subseteq E$, let's call its ratio of positive to negative examples q, the number of positive examples p and the number of negative examples n. [2] Then q is defined as:

$$q = \frac{p}{p+n}$$

The entropy (denoted H) of S is then defined as:

$$H(S) = -\Big(q \cdot \log_2 q + (1-q) \cdot \log_2 (1-q)\Big).$$

Important note: Normally, $\log_2 0 = -\infty$ but the ID3 algorithm uses $\log_2 0 = 0$ in its calculations! So you **MUST** treat $\log_2 0$ as 0 in your entropy calculations.

Remainder. Now that we know how to calculate the entropy of a set of examples, we can calculate the remainder and the information gain of an *attribute*. The remainder of an attribute with respect to a set of examples E is computed by first using the attribute to split the set of examples E into distinct subsets of examples, where each subset has a particular value for that attribute. The remainder is then a weighted sum of the entropies of the subsets; where a subset's weight is its proportion (in size) with respect to the total set of examples. More precisely, for some attribute Att that can take on d possible values

[1] Entropy is a concept originally from Physics that was adapted by Claude Shannon to quantify the amount of *information* in a process. Here it is used to quantify homogeneity but we could also interpret homogeneity as information.

[2] Remember that it doesn't matter which of the two classifications is called positive and which is called negative. These are simply ways to distinguish between one classification and the other. Therefore, in your code it is OK to arbitrarily choose one classification as positive and the other as negative.

and that creates a split that includes the subset of examples $S_1 \subseteq E$ through $S_d \subseteq E$, the remainder of Att is:

$$R(\text{Att}) = \sum_{i=1}^{d} \frac{|S_i|}{|E|} \cdot H(S_i),$$

where $|S_i|$ and $|E|$ are the sizes of the sets S_i and E, respectively. Let's see an example. Suppose the attribute we are considering is the Pat attribute. Then, as above, the split is $S_N = \{7, 11\}$, $S_S = \{1, 3, 6, 8\}$ and $S_F = \{2, 4, 5, 9, 10, 12\}$. The remainder of Pat would then be

$$R(\text{Pat}) = \frac{1}{6} \cdot H(S_N) + \frac{1}{3} \cdot H(S_S) + \frac{1}{2} \cdot H(S_F)$$

Information gain. We are now ready to see how to compute the information gain of an *attribute*. As mentioned above, the information gain is how we quantify the importance of an attribute. The information gain of an attribute Att with respect to a set of examples E is the difference between the entropy of E and the remainder of Att. More precisely, it is defined as

$$\mathsf{Gain}(\text{Att}) = H(E) - R(\text{Att}).$$

As mentioned above, an attribute' s importance is measured by its information gain. So in the ID3 algorithm, when we need to choose an attribute for a particular node we compute the information gain of all the attributes and choose the one with the highest information gain. We then store it in the current node, split the examples using the attribute, and recur on each of the subsets generated by the split (in the recursive calls we omit the current attribute since we've already used it).

8 On Training and Testing Data

Generally, when you create a machine learning model (in this case a decision tree) you use your training data to train the model (i.e., to create the decision tree) and also to test it. *It is crucial, however, that you not use the same parts of your dataset to train and test.* The data must be split into two non-overlapping parts: one for training and one for testing.

The training data is the data you will use to build and train your tree. This is the data that your tree will "learn" from. Given these training examples, your tree will learn how to classify examples.

Once you have trained your tree, you will want to test to make sure it is able to make accurate classifications on new examples (i.e., examples that it has never seen during training). This is where the testing dataset comes in. Your testing data will contain examples that were not a part of your training data and can be used to check the accuracy of your tree and how well your tree behaves when it encounters examples it has not seen before.

When you run your program, click 'Train' to train the tree on a dataset, and 'Test' to test it. You'll be prompted to pick a data file for each. Use the data in:

`/course/cs0160/lib/decisiontree-data`.

The visualizer will show you the tree your algorithm produces and give you feedback on what percent of the new examples it classifies correctly.

We've provided three data sets for you: `short-data`, which is our restaurant example from lecture; `villain`, which is medium-sized (around 200 examples); and `mushrooms`, which has several thousand examples and will produce a more complex tree. The decision being made for the `villain` dataset is whether a given person is a hero or a villain based on their opinions on food options at Brown; for `mushrooms` it is whether a mushroom is edible or poisonous based on physical characteristics. We recommend getting your tree producing correct results (i.e., matching the demo) for `short-data` before moving on to other datasets.

For each dataset, there are three files: `full`, `training`, and `testing`. `full` is all of the data we have. `training` contains a subset of `full` (about 3/4 of it). `testing` contains the other 1/4. The standard practice is to train on `training` and test on `testing`, of course, but we encourage you to try other combinations and compare them to the demo to see how your tree does. For example, when trained on `full` and tested on either subset, your tree should be 100 percent accurate.

8.1 Testing Data and Grading

You should be using the demo as a benchmark to determine what accuracy percentage your tree should achieve and what your trees should look like. Note that an accuracy percentage that is near what the demo gets does not mean your algorithm is correct, and if your tree looks different from the demo, it's probably an indication of a bug. If your tree gets the same or nearly the same accuracy at the demo, but is larger (splits more times), it is incorrect, because the optimal decision tree is the smallest possible tree that makes the correct decision.

Your grade will be based on a combination of your accuracy percentages, whether the trees you produce match the demo, how correct your code is, and code factoring, design, and style.

8.2 Testing data vs. JUnit testing

In short, testing data is a set of data that you can use to test the *overall accuracy* of your tree once you have created it using the training data. Testing data contains data points or "situations" that are not included in your training data so that you can see how well your tree behaves in situations that it has not been trained on. Once you are finished creating your tree, you can use the visualizer to see how well it performs with the testing data.

JUnit testing on the other hand, is probably something that you are more familiar with after Heap. JUnit tests should be used to check that each individual part of your program is working as it should. While you are coding, you can (and should!) factor out your code into helper methods so that each of these can be tested in your JUnit tests. We recommend that you do JUnit testing on your helper methods as you code them so that you're sure they work before using them in other parts of your algorithm.

In general, you should make use of the testing data to test the overall accuracy of your tree once it has been created and write JUnit tests to check that the smaller components of your program (your helper methods) are working as they should.

9 Important Notes and Reminders

- Your implementation **MUST** be recursive.
- The ID3 algorithm treats $\log_2 0$ as 0 not as $-\infty$ (or in Java, not a number: NaN). When calculating entropy, you must have a case that hard-codes $\log_2 0$ to result in 0.
- You will be doing string comparisons to find the names of values, attributes and classifications. Strings in Java are objects, not primitive types like integers, chars, or booleans. That means that every string is a distinct instance of String. Comparing two strings to see if they have the same value like "Foo" == "Foo" will never result in true since Java sees them as distinct objects (Unless you are comparing the same string instance to itself). The correct way to compare strings in Java is "Foo".equals("Foo"), which compares the actual values of the string. Make sure to be aware of this since it is a tricky bug!
- Beware of floating point arithmetic. In integer division, all values greater than 0 and less than 1 will be truncated to 0. Make sure you do not truncate these values when you don't want to. One way to prevent problems might be to store variables as doubles, evening if they're keeping track of an integer value such as a count, so that they do not get truncated when you divide. Looking for this is a good place to start debugging!
- Beware of `NaN`, meaning Not a Number. In particular, be aware that if you divide by the double 0.0, Java will not throw an exception, as it would with integers, but the computation's result will be `NaN`. This bug will produce incorrect trees that do not match the demo. Watch out for this in your remainder and entropy calculations; you need to account for it in the case where the number of examples with a particular value for an attribute is 0.
- Factoring your code effectively and creating helper methods is a crucial part of this project, and will be a part of your grade. Before you start coding, consider the

algorithm and the math that goes behind it from a high level and think about what helper methods you should make. You should aim to have as little repeated code as possible in your algorithm. This will make your program much easier to debug and is good design!

- To implement some of the formulas used for finding attribute importance, we recommend you use the Java Math package (link).

10 Support Code

We have provided support code for this project in the form of a tree visualizer and a number of classes that you will need to understand and use when writing your algorithm. Refer to the Javadocs (link) for more information. Understanding the format in which the data is given to you is crucial.

11 Working Locally

As always, you are welcome to work on this project locally, on your own machine and not over SSH or FastX, but you are by no means required to do so. To work locally, you will need to both set up Eclipse locally, and install GraphViz, an external library required by this project.

Important: The code you hand in must work on a department machine using the `make run` script detailed in Section 1. Because you *will* lose points if your code does not run on a department machine, we highly recommend testing your code using this script before handing it in.

11.1 Eclipse

If you choose to use Eclipse locally, you can follow the same steps in the Using Eclipse section. Just make sure to have the .jar files we require for this course (listed above) stored somewhere on your local machine.

11.2 GraphViz

The support code and tree visualization for this project make use of GraphViz, an external library for visualizing graphs and trees. In order to use our visualizer locally, you must install GraphViz on your computer.

Because the setup for GraphViz is somewhat involved, we have written detailed instructions in another document which you can find here (link). It is very important that you follow this document exactly, because the interactions between our support code and GraphViz are easily broken, and difficult to debug. If you encounter any problems setting up GraphViz, feel free to come to hours or work on a department machine.

12 What to Hand In

1. A filled in and commented `MyID3` and `MyID3Test` class and the unchanged `App` and `TestRunner.java` classes. Please also make sure to leave your Makefile in the same directory as your Java files.
2. Any additional classes you may have written.
3. A README named `README.txt` (see the README Guide for help).

13 Sources

1. Russell, Stuart J, and Peter Norvig. *Artificial Intelligence: A Modern Approach*. 3rd ed., Pearson Education, Inc., 2010.