CS158 - Assignment 2
Decision Trees

Due: Sunday January 30, 11:59pm

For our first programming assignment, we will be writing code that constructs decision trees based on training data. Make sure to read through the entire handout *before* starting, since I give some hints, etc. later on. You may (and I would strongly encourage you to) work with a partner on this assignment. If you do, you must both be there whenever you are working on the assignment. If you'd like help finding a partner, e-mail me asap and I can try and pair people up.

1 High-level Requirements

Implement the top-down decision tree learning algorithm we discussed in class. Your basic algorithm:

- Should handle base cases 1-4 as discussed on the "Final DT algorithm" slide, i.e. all stopping criteria except those to avoid overfitting.

- Should use minimum training error after splitting as the criterion for picking features. In the case of a tie, break the tie based on feature number with lower feature numbers being better (i.e. more likely to split).
- Need only work for binary features. For our framework, they will still be represented as `double`s but you can assume all features will either have value 0.0 or 1.0.
- For consistency, the left branch of the tree should correspond to feature value 0.0 and the right branch to 1.0.
- The learning algorithm should also support early stopping based on the depth of the tree.

2 Starter Code

I have provided you with some starter code that handles some of the more mundane tasks like data handling as well as a start for some of the class you'll implement. You can download it at:

`http://www.cs.pomona.edu/classes/cs158/assignments/assign2/assign2-starter.zip`

There are two directories in the starter, `code`, which has the starter code and, `data`, which has some different data sets to possibly play with for the dataset. The main dataset we'll be playing with is the titanic dataset that we played with last time (i.e., `titanic-train.csv`).

To get started look through all of this code and see what's available to you already.

Here is a quick summary of the classes I have provided to you:

- `Example`: The `Example` class is used to store an individual example from our data set. The three most common methods you'll likely use are `getFeatureSet`, `getFeature` and `getLabel`.
- `DataSet`: A representation for a data set which consists of a collection of examples along with some meta information. To create a new data set object for the Titanic data set:

  ```
  DataSet dataset = new DataSet("train-titanic.csv")
  ```

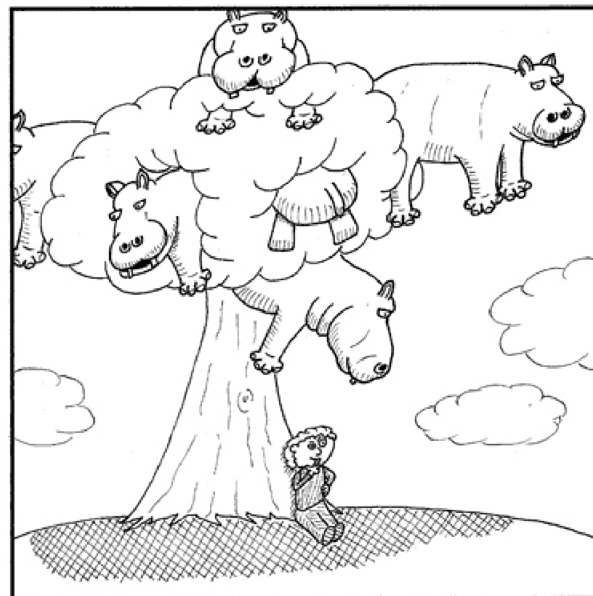
  You'll likely need to use all of the functions in the class, so make sure you understand what each of them do.
- `CSVDataReader`: This is a helper class for the `DataSet` class and you shouldn't have to use this explicitly.
- `Classifier`: An interface for classifiers. The class you're writing will implement this.
- `DecisionTreeNode`: You don't have to use this class if you'd rather write your own, but this class represents the nodes of a decision tree. It can be used to represent both internal nodes as well as leaf nodes and has various support for accessing data, setting attributes and for printing out the final tree.

To use the `treeString` method of this class, you need to pass it the feature mapping information from the data set (assuming you want to see the feature names). So, if you had a `DecisionTreeNode` called `root` that was the top of your trained tree and a `DataSet` object you can print out the entire tree with feature names using:

```
System.out.println(root.treeString(dataset.getFeatureMap()));
```

You **may not** modify any of: `Data`, `DataSet`, `CSVDataReader` or `Classifier`. If for some reason you think you need to, please come talk to me. Also, please use the *package* structure I have provided (i.e. `ml` and `ml.classifiers`). If you need a refresher on packages, please come talk to me.

3 The Nitty Gritty



It was, actually, under this hippo tree where Isaac Newton's fierce physicist rival, Bernard Johns, would soon *first* discover the theory of gravity. *...however...*

You must write a class called `DecisionTreeClassifer` in the `ml.classifiers` package that meets the following specifications:

- Implements the `Classifier` interface.
- Has a zero parameter constructor.

- Has a method:

  ```
  void setDepthLimit(int depth)
  ```

  that takes one parameter and sets the limit of the depth of the tree that will be generated when training. A depth one cutoff would produce a tree that has one internal node and two leaf nodes (what we've called a "decision stump") and a depth zero cutoff would produce just a single leaf node that predicts the majority label.

- Has a `toString` method that prints out the decision tree. If you decide to use the `DecisionTreeNode` class, you can simply call the `treeString` method on the root of your decision tree and return that string.

- The `train` method must:

  1. Train the model based on the dataset following the specifications from Section 1.
  2. Take the depth limit restriction into account. If the depth limit has *not* been set, the entire tree should be learned. If the depth limit has been set, only nodes up to that depth limit should be built.

- The `classify` method classifies the example (i.e. predicts the label) based on the learned tree.

4 Evaluation

Once you have your classifier working we now want to see how well it does and examine some of the features. Create a class called `Experimenter` in the `ml` package and put your code for this section in that class.

Answer the following questions about the `titanic-train.csv` dataset (**not** the real-valued version!) and put them in a file called `experiments` (pick some reasonable file type). Explicitly label each answer with the question number. At the top of this file, but your name(s).

1. What is the accuracy of the **random** classifier on the Titanic data set from assignment 1. To calculate this, generate a random 80/20 split (using `dataset.split(0.8)`) train the model on the 80% fraction and then evaluate the accuracy on the 20% fraction. Repeat this 100 times and average the result (hint: do the repetition in code :).

2. What is the accuracy of your decision tree classifier on the Titanic data set with unlimited depth. As above, average the results over 100 random 80/20 splits.

3. What is the best depth limit to use for this data? To answer this, do the same calculations as above (average 100 experiments), but do it for increasing depth limits, specifically 0, 1, 2, ..., 10. Show all of your results.

4. Do we see overfitting with this data set? Repeat the experiment from question 3 with increasing depth (0, 1, ..., 10) and calculate the accuracy this time on both the testing data (like before) *and* the training data. Create a graph with these results and then provide a 1-2 sentence answer describing the graph.

5. How does the amount of training data affect performance? To answer this, do the same calculations as above (average 100 experiments), but start with splits of 0.05 (5% of the data used for training) and work up to splits of size 0.9 (90% of the data used for training) in increments of 0.05. For these experiments use full depth trees, i.e. trees without any depth limit. Create a graph with these results and then provide a 1-2 sentence answer describing the graph.
6. What does the training data size experiment tell us?

5 Hints/Advice

- In addition to the Titanic data, I have also included a slightly modified version of the data in Figure 4.6 from the Tan et al. book (which I'm sure you've all read at this point...) in a file called `default.csv` in the data part of the starter.

  Work through this example by hand, calculating all of the errors, etc. at each step. You can then use this to check to make sure that your algorithm is doing the right thing.

  I promise you spending a bit of time upfront will save you a lot of debugging time!

- Debugging the training algorithm is tricky. The easiest way to avoid hard-to-find bugs is to build your program up incrementally, testing as you go. Try hard to break your training procedure into multiple procedures and test each of these procedures to make sure they work individually. For example, you might write functions that split a given data set around a particular feature and that calculate the training error of a split.
- The train method only takes a `DataSet` as a parameter and doesn't return anything. To setup a recursive solution properly, you'll probably want more parameters and you'll definitely want to return something (probably a `DecisionTreeNode`). To do this, you'll need to write a helper function that will do all the dirty work. Then, call this helper function from your train method.
- As you recursively build the decision tree you'll need to keep track of either the features left or the features used. If you use some sort of set (which is very reasonable) you'll need to make sure to *make a copy* each time you move to the next level of the tree, i.e. splitting the data on a feature and then recursing. If you don't do this, you'll be sharing a single object across all calls, which is NOT the intended behavior you'll want. Make sure that you understand this idea!
- Start with the most basic version of the training algorithm, e.g. just one base case stopping when the data consists of a majority class. Once you have this working, go back in and add the other base cases, including the depth limit.
- As you add in the additional base cases, modify the Tan et al. example to include each of the base cases and check that it does the right thing. The Titanic data set doesn't necessarily have all of the base cases in it, however, I will be checking them all :)

- This should not be a lot of code, but make sure you think very clearly about how everything will fit together. If you find yourself doing weird/complicated things, there's probably an easier way to write it.

6 Extra Credit

For those who would like to experiment (and push themselves) a bit more with decision trees (and of course, get a bit of extra credit) you can try out some of these extra credit options. If you try out any of these option save a full copy of your basic working code *first*. Many of these won't change the functionality, but just in case, it's good to have a backup.

- Implement one of the other early stopping criteria (or pruning techniques). If you go this route, include a comparison of the performance of this criteria vs. the depth limit criteria in your writeup.
- Implement one of the other splitting criteria that we talked about in class (either entropy or gini). If you go this route, include a comparison of the performance of this splitting criteria vs. training error in your writeup.
- Add functionality to support non-binary features. In `titanic-train.real_valued.csv` I've included another version of the Titanic data set with all real valued features and a few additional features. These are the exact same examples so you can compare your performance to your old results. If you go this route, compare the results from the binary-only decision tree to the non-binary decision tree in your writeup.

7 When You're Done

Make sure that your code compiles, that your files are named as specified and that you have followed the specifications exactly (i.e. method names, number of parameters, etc.).

Create a directory with your last name, followed by the assignment number, for example, for this assignment mine would be `kauchak2`. If you worked with a partner, put both last names.

Inside this directory, create a `code` directory and copy all of your code into this directory, maintaining the package structure.

Finally, also include your `experiments` file with the answers/graphs from Section 4.

`zip` this folder and submit that file through sakai. If you worked with a partner, only one of you needs to submit it, but make sure both your names on it!

Commenting and code style

Your code should be commented appropriately (though you don't need to go overboard). The most important things:

- Your name (or names) and the assignment number should be at the top of each file
- Each class and method should have an appropriate doc comment[1]. All of the code I gave you already has appropriate doc comments and if you use Eclipse or some similar package, it will do a lot of this for you already.
- If anything is complicated, it should include some comments.

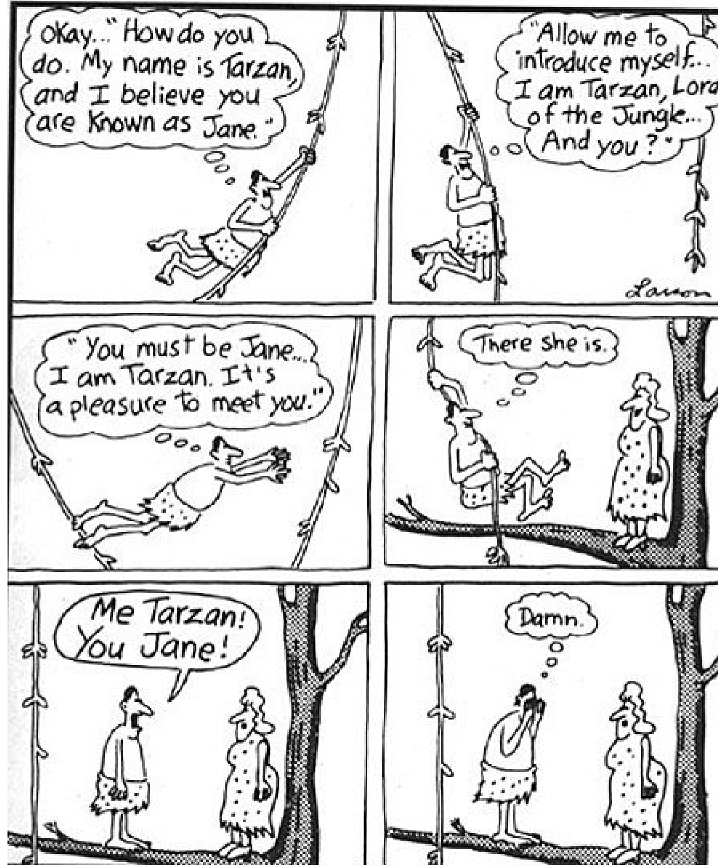
There are many possible ways to approach this problem, which makes code style and comments very important here so that I can understand what you did. For this reason, you will lose points for poorly commented or poorly organized code.

Grading

	points
Correction & functionality	31
Syle/comments	4
Writeup	10
total	45

---

[1] See `http://www.oracle.com/technetwork/articles/java/index-137868.html` for details on this if you're rusty.



more awesome pictures at THEMETAPICTURE.COM