Decision Trees Pt. 2: Training & Testing

Eckel, TJHSST AI2, Spring 2021

Background & Explanation

In the prior lab, we created a perfect decision tree for a certain set of feature vector / classification pairs. For some data sets, like the mushrooms, this is exactly what we want. For others, it isn't; often, in fact usually, the purpose of machine learning like this is to use a certain *training set* to create a classification network, and then use it to classify *new feature vectors* that weren't in the training set. We aren't getting to this level of sophistication yet, but as an example, we may train a network to distinguish dogs and cats, and then we want to be able to send it new pictures of dogs and cats that weren't in the training set and have it distinguish those correctly.

So, in this assignment, we'll take a look at some larger, messier data sets and experiment with using a subset as a training set and then testing its accuracy on a test set from the remaining data that wasn't used for training. In doing so, we'll build an understanding of when creating a decision tree makes sense for the data and when it doesn't.

Get pyplot / Learn .scatter()

For this assignment, you'll need to make scatterplots using pyplot, which you can import from the matplotlib library. By now you should know how to use pip or a package manager to install matplotlib; this is another one that comes by default on Anaconda's base environment, too. As for using pyplot to make a scatter plot, I leave learning this up to you - there are many online tutorials! A few things I want to be sure are clear, though:

- You won't need to import numpy for anything I'm asking you to do on this assignment (though for those of you curious, we will learn about numpy next unit.)
- Spyder will show the plots you create automatically, but for my grader to work you need to actually call ".show()" after you create a plot; if this doesn't appear in the online tutorials you find, let me know.
- I'm not looking for you to use any fancy features like data point coloring. But: adding x-axis and y-axis labels would be nice!

To make sure you're ready to use pyplot, you should probably do something like make a list of 100 random x values and a list of 100 random y values and then pass them to .scatter() and make a scatterplot with axis labels.

Once you've done that, we're ready to get back to decision trees!

Measuring Accuracy with a Test Set

To judge the validity of any machine learning model, typically some observations are used to train the model and a separate, distinct set of observations is used to test it. These are data points that were not used in creating the model, but are pre-classified correctly so we can check our model's accuracy. They should give a good representation of how the model can be used productively on new data.

Specifically, we'll measure the **percentage accuracy** on the test set. The test set, like the training set, will come with a classification for each observation. We'll run each observation's inputs through our model and see how the model classifies it, keeping track of how many observations are classified correctly. The accuracy of our model on this test set will be the percentage of observations classified correctly, as you might have guessed.

Creating a Training Set & Test Set

For this task, you'll be using the house-votes-84 data set, voting records on several bills of republicans and democrats in congress. First, note that each row of data has a label at index 0; **you'll need to discard these labels**, they aren't valid features to differentiate on. You may also notice that many of the feature vectors are missing features – that is, some of the representatives didn't vote on particular bills. We'll start by handling this in an easy way.

1) From the original data set, drop all rows that contain a "?". This data set, which we'll call NONMISSING, should have 232 rows.
2) Remove the **last 50 rows**; call this the test set. Do not use any of these rows for training.
3) Now, build a learning curve.
    Loop a SIZE variable from 5 to 182.
        Randomly select SIZE rows from NONMISSING that are NOT in TEST; this is TRAIN.
            Ensure that not every data point in TRAIN has the same classification; if so, re-select.
        Build a tree using TRAIN
        For each feature vector in TEST
            Use your tree to classify the feature vector
            Compare with that vector's actual classification
            Keep track of how many are classified correctly
        Calculate the percentage accuracy of your tree
        Store (SIZE, accuracy) in a list
    Using pyplot, plot a scatterplot of training set size vs accuracy

After you've done this successfully, screenshot your plot and **send it to me on Mattermost**.

One interesting thing that you might notice is that there are smaller training sets that produce better results than the full 182; this is called **overfitting**, when you make a model so precise to your initial data that it actually does worse on new input. If you do see overfitting here, though, it's likely to be pretty mild; we'll talk more about this concept in later lessons.

For now, what I want you to notice is how quickly the trained tree becomes quite good!

Dealing with Bad Data

Ok: let's return to this problem of the "?"s. We want to be able to include every data point in our training / testing process! So: let's fill in our missing data values with our best guess. This is one way of dealing with missing data; you may invent another as part of your OW credit (see the end of the document). But for now let's do this:

1) For each independent variable, see how many republicans voted yes or no, and see how many democrats voted yes or no. Then, loop over each observation in the data set, and for any "?" values, replace that with how the majority of that data point's classification voted. (ie, if you see a "?" for V5 for an observation classified as "republican", look up whether more republicans voted "y" or "n" for V5 and replace the "?" with whichever one was more common.)
2) If you do this correctly, this will result in at least one instance where two vectors with exactly the same voting record are classified differently, which is a pretty common occurrence in machine learning; the real world is messy. You'll need to add something into your tree generator to deal with this. In particular, if the entropy is nonzero but every single choice of feature produces an information gain of zero, you're in this situation. In this case, randomly pick one of the vectors' classification and set that as the classification of this branch in the tree.
3) Repeat the previous task using this new data set with once again the last 50 observations being used as a test set. Loop SIZE from 5 to 385 this time, since we're now using all 435 observations in the data set. Generate your scatterplot again. Do you see any notable differences in the scatterplots when using only complete data vs our majority-rules-gap-filled data? **Send me a message on Mattermost** with your new plot and your thoughts!

What About Incomplete Trees?

There's a problem that you're going to encounter in the next part of this assignment, so, we need a brief digression before we continue.

In the voting data set, each feature only had two values – Y or N. So if a given feature appears in the decision tree anywhere, it must branch into at least two options… and there ARE only two options… so every option must be present. But in the next two data sets, some features have multiple possible values. Therefore, it may be true that, down a certain path of your tree, a feature appears and the input vectors that have made it to that point only contain SOME values of that feature, not all of them. So, your tree will have a split on a certain feature that doesn't contain all values of that feature. And then when you run your test set there might be another input vector that follows that same path up to that same split, but it has a value for that feature that isn't in the tree. (In other words, the training set may not have every possible combination that appears in the test set, and since trees are purely deterministic, any missing combination may result in a tree that doesn't know where to go.)

So: how to address this problem?

Well, first and foremost, if a particular combination didn't occur anywhere in a large training set, it's likely to be pretty uncommon. So it doesn't matter too much to our final accuracy how we decide to address this problem, because it's only going to apply to a small number of cases. So it is entirely fine to check and see if the current input vector's value for the current feature appears in the tree, and if not, just **choose a random branch**.

If you want to do a little better, you can **go down all branches** and then go through the rest of the tree as normal, getting multiple possible classifications, and then choosing whichever classification is given most often. You can even **weight the branches** according to how many different training set input vectors were classified along each path if you like, producing a probabilistic average. But this is all bonus; it's really fine to just choose a random branch.

The actual way that this problem is best handled is with a **random forest**, which is an OW at the end of this assignment; see more details there. But either way, the impact will be minimal and it won't change the broad conclusions of the next section.

With that established, let's move on!

Advantages and Disadvantages of Decision Trees

We'll finish with a section that I suspect you won't find surprising – as it turns out, decision trees are very good at modeling situations where something like a decision tree was actually used, and not so good at modeling situations that result from extremely different processes. But let's put some numbers to this and see just how true this is.

You have two other large data sets – nursery.csv and connect-four.csv. The nursery data set is from nursery school admissions in Ljubljana, Slovenia in the 1980s. (And you thought machine learning wouldn't show you anything cool, didn't you?) Apparently, nursery school demand exploded and so a strict admissions flowchart was developed. So you might expect a decision tree learning model to do very well! The connect four data set contains every possible state of the connect four board after 8 moves where neither player has won and the next move is not forced; the classification is the expected winner of each board with perfect play. This is a situation more reliant on spatial patterns, so you would expect a decision tree to have a harder time.

Neither of these data sets contains any missing values; neither of these data sets has an index 0 label.

Let's go ahead and experiment:

- For nursery.csv, take a random 500 rows and make them the test set. (Possibly: shuffle the entire data set, and then take the last 500?) The remaining data contains just over 12000 rows. Loop from a training set size of 500

up to 12000 with a step of 500 (ie, try 500 then 1000 then 1500, etc) and plot the size vs test set accuracy on pyplot.
- For connect-four.csv, we have even more data points to work with. Take a random 7000 rows and make them the test set. The remaining data set contains just over 60000 data points. Loop from a training set size of 5000 up to 60000 with a step of 5000 and plot the size vs test set accuracy on pyplot.
- **Send me a Mattermost message** with both plots and your analysis. How good does each decision tree model get? Does this match what you'd expect given the situations each data set represents?

Get Your Code Ready to Turn In

I'm going to run your code on a new data set you haven't seen. Some things to know about this data set:

- There are no missing values.
- There is no index column to discard.
- There are more than two possible classifications.

You'll get command line inputs in this form:

```
yourfile.py filename.csv 40 30 120 10
```

These represent, in order, the data file, the size of the test set, the min training set size, the max training set size, and the step. (In other words, in this case you'd make a loop from 30 to 120 with a step of 10 and try a training set of each size). I won't pull any shenanigans; the length of the file will be greater than test set size + max train set size.

Your code will need to read in the data file, randomly select the given number of observations to place in a test set, and then run the specified training loop for various training set sizes. It will then need to graph training set size vs accuracy and **do not forget to call .show() to show your final scatterplot**.

Specification for Decision Trees Part 2

Submit **a single python script** to the link on the course website.

This assignment is **complete** if:

- The "Name" field on the Dropbox submission form contains your **class period**, then your **last name**, then your **first name**, in that order.
- Your code accepts command line arguments and generates a plot as described on the previous page.
- Runtime should be reasonable; I'm not too worried about this, but if you are, let me know. If your runtime is quick for the full republican/democrat data set you should be fine!

Specification for Outstanding Work: Non-Categorical Data

There's a very messy data set for you on the course website – weatherAUS.csv. The final column, as in all our data sets for this assignment, represents the classification. In this case, "will it rain tomorrow?" The goal, then, is to make a model that will predict if it rains tomorrow. (More information about this data is also linked on the course website.)

This data set has two possible classification columns; you'll want to pick one. The RainTomorrow column is a binary classifier, YES or NO. The previous column, RISK_MM, is the value of the rain tomorrow in mm. This can be used as a sort of continuous response variable, which might make training more accurate. You can use either column, but you can only use one of them; if you leave them both in, then one will predict the other and your model won't make any sense.

Once you choose which value you'll use for output, there are several problems to consider:

- The first column, date, should probably be discarded… or maybe converted into month? Could there be different things that predict rain on the next day in certain months? What do you think?
- The second column, location, is also complicated. Should that be included as a possible differentiating decision point or ignored? Or perhaps you want to make a model that is JUST for one specific location?
- There is a lot of missing data; here, missing values are given by "NA" instead of "?", and you'll need to deal with that somehow.
- Finally, and this is the big one – **much of this data is not categorical** like the data we've seen so far! So, in order to make this into a decision tree model, you'll need to decide on buckets in which to place the given values, ranges of values that constitute separate decisions on the tree. How many different ranges should there be? What should they contain? Is there some way to generate this based on the data as given?

This is a tough OW spec, but it's also an extremely authentic real world problem and a good experience with what working with real data is like. The goal is simple: **what model can you make from this data, and how good is it at predicting rain tomorrow?**

Write a concise write-up that answers these questions:

1) What did you do with "Date", if anything?
2) What did you do with "Location", if anything?
3) How did you deal with missing data?
4) How did you deal with numerical data?
5) How big were your test / train sets? Why did you choose those sizes?
6) What were your results? Be specific!

Submit your python file and write-up to the link on the course website.

Specification for Outstanding Work: Random Forest Classifiers

This is a new idea this year so this one isn't especially fully formed; this is a good choice if you're curious to investigate something further and give me advice on how useful/interesting it is.

To avoid overfitting and to avoid a particular tree not containing a particular combination, in practice decision trees are often made in batches using a random forest approach. In simplest form, this means that you randomly sample – with replacement – multiple smaller subsets from the whole large training set and use each one to make a decision tree. Then, each element of the test set runs through all the generated decision trees, and the mode of the classifications is taken as the data point's actual classification.

I haven't yet tried to implement this on any of our data sets, but I'm quite curious about the concept. To get OW credit for this, write a random forest algorithm that generates, oh say, 10 trees or so, each with size less than a tenth of the full training set. Then try this version of classification compared with the results we got from just making one big tree. Try this on all the data sets – representatives, nursery, and connect four. Are there improvements to be found on any of them?

A further complication comes from what's called the random subspace method, which involves randomly sampling **features** when making each tree, making trees that individually respond to particular features more strongly by ignoring some of the others. This is most useful for things like fMRI data, where there are more features than data points, but I'd love for you to play around with that and see what happens on these data sets as well? This is optional, but interesting!

This doesn't have a submission link, because I'm not entirely sure what I'm asking for. Talk to me on Mattermost, tell me what you tried, send me some graphs or some data, and be prepared to maybe run a few follow-up questions?