

### ***Decision Trees and Random Tree Forests***

We have spent the last couple weeks looking at Neural Networks and how to use them to solve classification tasks. In particular, we looked at how to do optical character recognition (OCR) on hand-written digits using a one-, or two-layer Neural Net. The networks we build performed pretty well.

In general, Neural Nets can be designed to handle all kinds of classification tasks including recognizing particular types of objects or categories in pictures, speech recognition, medical diagnosis, and much more. Classification is an incredibly important, general category of problems that are often the focus of AI and machine learning techniques.

In this Unit, we will learn about a different framework for classification based on ***tree stumps***, and ***decision trees***.

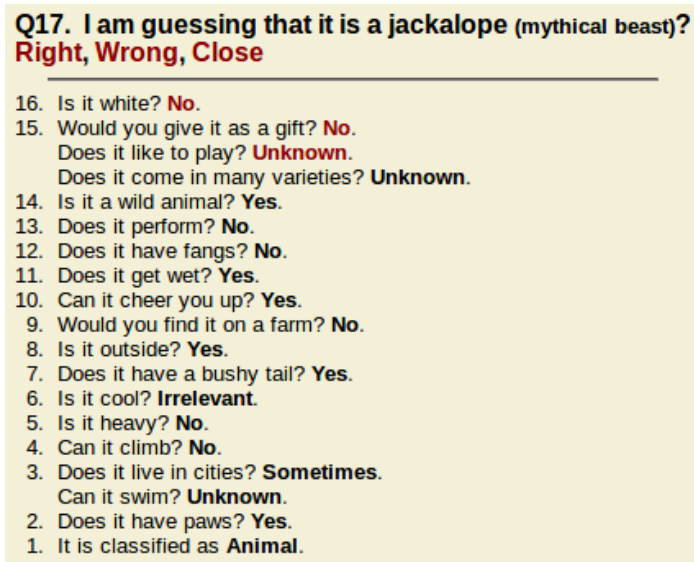
#### ***Decision Stumps***

In order to understand how decision trees work, you have to play a little game:

<http://www.20q.net/>

(You have to think of something, ***anything***, and the game is going to try to guess what you thought of by asking you 20 questions about it).

Here's what happened with my game:



**Q17. I am guessing that it is a jackalope (mythical beast)?**  
**Right, Wrong, Close**

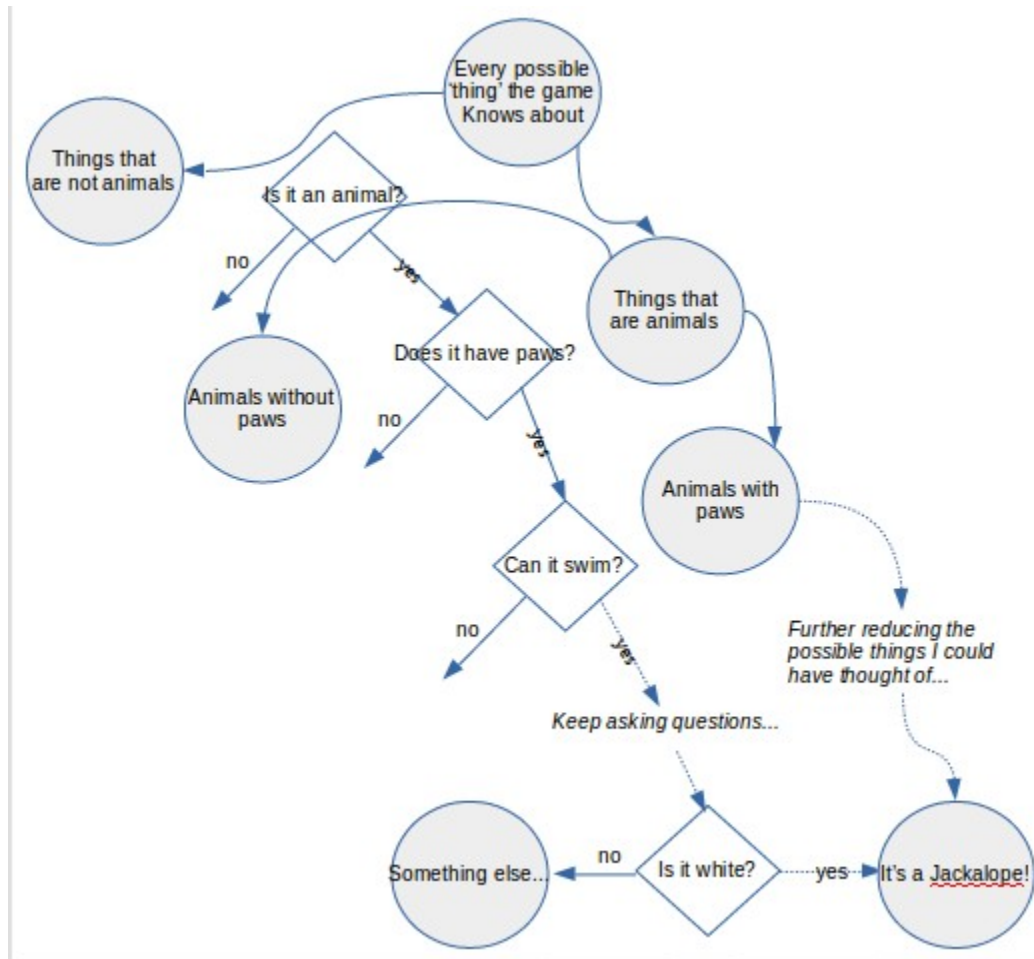
---

16. Is it white? **No.**
15. Would you give it as a gift? **No.**  
Does it like to play? **Unknown.**  
Does it come in many varieties? **Unknown.**
14. Is it a wild animal? **Yes.**
13. Does it perform? **No.**
12. Does it have fangs? **No.**
11. Does it get wet? **Yes.**
10. Can it cheer you up? **Yes.**
9. Would you find it on a farm? **No.**
8. Is it outside? **Yes.**
7. Does it have a bushy tail? **Yes.**
6. Is it cool? **Irrelevant.**
5. Is it heavy? **No.**
4. Can it climb? **No.**
3. Does it live in cities? **Sometimes.**  
Can it swim? **Unknown.**
2. Does it have paws? **Yes.**
1. It is classified as **Animal.**

(I actually thought of a ***marmot***)

Often, the thing guesses just right – and in the example above there are several questions to which I was not sure what the right thing to say would be for a marmot - ‘Can it cheer you up?’ - well, depends on whether you like marmots or not!

The point is, the way the game tries to decide what you're thinking of is by building a tree using the questions and answers you have provided, and using that tree to create subsets of objects that have the properties that agree with your answers to their questions, like so:



We start at the top, with a set of all possible things the game knows about – ***each of them is associated with a vector of all the properties or features that a thing can have, and their corresponding value.*** For example, properties or features may be things like ‘size’, ‘color’, ‘is an animal’, ‘has paws’, etc. And for each feature, we must provide the value that applies to that feature for each thing in our collection.

Some features will have binary values (yes/no), some may have a ***categorical value***, for example: ‘type of animal’ may be one of ‘mammal, fish, bird, insect’, and only these 4 values are valid. Other features may have a real value attached to it. For instance, we could have a ‘weight’ feature where the weight is a real-valued quantity.

The type of feature and its domain is not as interesting at this point. What matters is, we have a vector with all the values for all the features for each of the things in the collection.

Ok, so at the top we have a set with all the things we know about, and their feature values. Then we ask a question: ‘Is it an animal?’. This is a clever question to ask, because the result of it will split the set of all things into two smaller ones – one for things that are animals, and one for things that are not. Depending on what the player is looking for, we can now focus on a smaller set of possible things!

In the case above, we’re thinking of an animal – so the next question is ‘Does it have paws?’ - again, it’s a good question since it splits our current set of animals into large categories that are mutually exclusive – we answered ‘yes’, so we never have to think of animals like dolphins, fish, or birds.

The game continues asking questions, with the goal of reducing further and further the possible set of things the user may be thinking of, until, hopefully, before 20 questions have been asked, we have arrived at a single thing whose features agree with the values the user provided.

In the case above, if you pay attention, you’ll notice that only 16 answers I provided were informative, for the others I chose ‘unknown’ so these provided no information. The fact the game guessed wrong is likely the result of my answers to some of these questions not being what the game has stored for the feature values of a ‘marmot’.

Importantly, once the thing we want goes to the wrong subset in the process above, we’re toast, we’ll never find it!

**The process above is a decision tree.** The root at the top corresponds to the first question, each node further down the tree is attempting to classify things based on the result of the question attached to the node. Each node is called a **decision stump**.

Let’s see how we build a working decision tree.

### ***Features, decision stumps, and tests***

As noted above, we will assume we have access to a **labeled training set** consisting of all the items we’re attempting to classify, along with the correct classes for each of them.

Each of our data items will be characterized by a **feature vector**:

$$\vec{v}_i = \{x_1, x_2, x_3, \dots, x_k\}$$

So data item ***i*** has ***k*** features, all stacked in a vector. And we have the corresponding class  $y_i$

Each feature could be:

- Binary (yes/no, represented with a 0 or 1 in the feature vector)
- Categorical (e.g. for a ‘size’ feature we could have ‘small, medium, large’, represented as values 0, 1, and 2 in the feature vector. Any number of categories can exist)
- Real valued (e.g. ‘weight’)
- Integer valued (e.g. RGB values for colour, each would be a separate feature, with values in

[0,255]).

Note that pretty much any other features can be reduced to the types of features above.

### ***Creating a decision stump***

A decision stump consists of:

- A choice of feature(s)
- A test to be applied to these features
- Links to the tests that are the children of this node (2 or more depending on the test)

### ***About the tests***

- Tests can use one or more variables.
- Single variable tests can, for example
  - \* Check for specific values (yes/no tests, or categorical, e.g. 'is the size [small, medium, large]')
  - \* Thresholds for integer/real valued data (e.g. is the 'weight' < 50Kg.?)
  - \* Equality or difference – for integer valued features (e.g. is the 'red' colour channel == 255?)
- Multiple variable tests usually involve comparisons
  - \* Is  $\text{feature}_p < \text{feature}_q$ ?
  - \* Is  $\text{feature}_p == \text{feature}_q$ ?
 And so on...
- Tests can have two or more outcomes, and there will be one child node for each outcome.

The job of each decision stump, is to take a set of input data, and ***split it into two or more subsets*** based on the result of the test.

### ***Building a Decision Tree***

Building a tree requires us to generate a sequence of tests that split the data into smaller and smaller subsets until we can return a ***class*** for each of the input samples. The tree will have more or fewer tests along different branches, and how many tests are required will depend on the features available, the difficulty of the problem, and how big of a tree we're willing to build.

Keep in mind – the goal of the tree is to return a ***classification result***, for example, if we are solving the handwritten digit recognition problem, we want the tree to return the correct digit for each input image. In the procedure below, ***distribution of classes*** means a ***histogram*** of how often each of the possible classes occur within the dataset a tree stump is working with.

For the handwritten recognition problem, the root node is working on the entire input dataset, so we can expect each digit appears with a frequency of about 10%. But once we start splitting the data set by applying tests, we expect the subsets to have higher proportions of some classes. ***This is the information that eventually allows us to make a decision*** as to which class an input sample belongs to.

Build Decision Tree:

Initialize an empty tree

Compute distribution of classes for original input dataset

**Create a test for the root node**

Loop until **no more nodes left to process**:

- \* Split **current** set of inputs according to the test in the **current node**
- \* Compute the distribution of classes for each subset

For each subset resulting from the test:

- If the subset is empty, return **most common class from current node** as the correct class
- If all items in this subset have **the same feature values**, return **the most common class** within this subset as the correct class
- If all items in this subset have **the same class**, return **this class** as the correct answer
- Otherwise: Create a child node for this subset, choose a test, and go back to **loop** to process this new node as the current node, with the corresponding subset of input data

So basically, at each level in the tree, we take the corresponding subset of input data passed on from the parent node's test (or the whole set if we're at root), and apply a test to it.

The result of this test may be enough to tell us the class of any items that made it to this node (the three cases above where we can return a class value), or we find that there is still a subset of data for which we need to do more tests – in which case we create another test and keep splitting.

The process terminates where at each branch of the tree we can provide a class for whatever data made it down to the leaf node in that branch. **Technically** it's possible to build a tree that perfectly classifies each input data sample! **But** in practice this typically leads to very large and not very useful trees (more on this later), so it's not uncommon to also set a **maximum number of levels** for the tree, and once we reach that level, leaf nodes simply return **the most common class** for the training data samples in their input data subset as the correct class.

**So:** The basic process of building a decision tree is fairly simple – but of course, there's a lot of work that goes into making sure our tree is as good as we can make it.

### **Choosing the right test**

The choice of test at each level of the tree is critical – some tests, applied to an input set, will produce little or no information:

- Tests such that **all** the input data gets classified into **a single** subset – no splitting actually happened!
- Tests such that **the class distributions for the subsets** are pretty much the same as the **class distribution for the current set**. This means this test didn't help us narrow down the possible classes for the current set of samples

And of course there is a range – a test such that **all but one** sample go to one of the subsets is just as bad as the test in which all the data goes to one subset. A test such that the resulting class distributions are just **a bit different** from the class distributions of the current set is not very useful.

So, we need a way to select a test that is going to help us classify the data we have with fewer stumps and a smaller tree.

### **Entropy**

The key idea regarding how we will choose a test is that we want to **reduce the uncertainty** regarding what is the correct class for samples in the subsets resulting from applying that test. We want to arrive at a point where, ideally, there is **no uncertainty** and **we know for sure** what the class should be.

The measure of uncertainty we use is called **entropy**:

$$H(S) = - \sum_i p(\text{class} = i | S) \log_2 p(\text{class} = i | S)$$

Here  $S$  is a subset of the input data, and  $p(\text{class}=i | S)$  is the frequency in the histogram of classes for this subsets for class  $i$ . For any given data set, the value of  $H(S)$  is greatest if the probabilities  $p(\text{class}=i|S)$  are all the same (you have a uniform distribution, complete uncertainty regarding which class a sample belongs to).

$H(S)$  decreases as the class distribution becomes less and less uniform.

### **Using entropy to select a good test**

Suppose we're given some subset of data  $S$ , and are asked to select a **good test** to use to split this dataset in a way that minimizes uncertainty. Given a **candidate test**, we can estimate **how good this test is at reducing uncertainty by computing**:

$H(S)$  - The entropy for the current dataset we're being asked to split

$H_j(S_j)$  - The entropy for **each of the subsets** resulting from the splits created by the candidate test

We can estimate **the expected entropy** in the subsets resulting from this candidate test's splits as:

$$E[H_j(S_j)] = \sum_j p(S_j) H_j(S_j)$$

This is just the weighted average of the entropy of the resulting subsets of data,  $p(S_j)$  is nothing more than the **percentage** of data in  $S$  that ends up in subset  $S_j$ . Given  $H_j(S_j)$  we can estimate the **reduction in uncertainty** as a result of applying the candidate test to the input data set  $S$ .

$$R(S) = H(S) - E[H_j(S_j)]$$

This is a measure of **how good the candidate test** is at helping us classify the data in set  $S$ .

So, in order to select a test for any node in the tree, we use the following process:

***select test:***

```
Given input data set S
Class distribution for set S
Entropy  $H(S)$ 

for i=1 to K           // Try out K possible candidate sets
    * Choose a random feature or set of features to test
    * Choose a random test that applies to the selected features
    * Evaluate  $R(S)$  for this test

Select from the K trials the test that has the largest  $R(S)$ 
```

So we simply try out a number of possible tests at random (how many depends on how much time we want to spend in building the tree), and choose amongst them the one that has the largest reduction in uncertainty.

Applying this process at each level of the tree results in a decision tree that is reasonably short and for which each node is doing a reasonably job in terms of helping you classify the subset of data it has to take care of. There is just one more detail that we need to keep in mind.

### ***Overfitting***

If we're not careful, and we just keep splitting nodes in the tree until all of them are terminal, we end up with a possibly large tree that is very good at classifying the input training set, but will not do so well on new data – which is after all what we want the tree for. This is called ***overfitting*** and we already talked about it in the context of neural nets. Decision trees are prone to overfitting, so it makes sense to put a limit on the height of the tree, and we often use heuristics to stop splitting after, for example, the resulting subset of data has fewer samples than some minimum threshold  $t$ .

We will shortly see a much stronger classification method that reduces significantly the overfitting problem while providing us with strong classification power on large datasets.

But before closing this part on decision trees, have a look at these two external references:

A ***beautiful*** description/animation of the process of building a decision tree:

<http://www.r2d3.us/visual-intro-to-machine-learning-part-1/>

Visualizations of decision tree building for different datasets:

<https://explained.ai/decision-tree-viz/>

### ***Randomized Forests***

Having a tree is nice, but having a whole forest! Well that's a nicer idea by a long shot.

As we noted above, a single decision tree can easily overfit the data, this together with the randomness of the process of building the tree means we're likely working with a sub-optimal tree for whatever problem we're attempting to solve.

However, we can improve things by **boosting** - the idea of boosting is that we can take a set of **weak-classifiers**, that is, individual classifiers that are not super good at the classification task they're trying to solve, and combine their predictions in some way to build a **strong classifier**.

In a way, a single decision tree is a mini example of **boosting** – it takes individual **decision stumps** which are weak classifiers, and builds a much stronger classifier by chaining them together along tree branches.

We can take the same idea and apply it to whole trees: We can build a strong classifier by training **many individual decision trees** and then combining their predictions to build our strong classifier.

The process is remarkably simple:

#### ***Building a Decision Tree Forest:***

Initialize empty forest

For each of K trees in our random forest

Select a random subset of the input training set

Train a *short* decision tree using this random subset of data

#### ***Classification with a Randomized Tree Forest***

Given a *test sample*

Obtain a classification result from each of the K trees in the forest

Return the class most frequently picked (majority vote)

A couple of things need explanation here:

- Building a *short* tree means we stop the process after a certain target height has been reached, this means each individual tree won't be as good as a fully trained decision tree that we let grow as much as needed, but it *doesn't have to be, it just has to be ok at classifying!*
- Selecting a random subset of training data is essential, it means *each tree has seen a different subset of training samples*, so there is little chance it can overfit the entire training set, and the resulting tree forest will be very resilient to overfitting

The problem, of course, now becomes figuring out what are reasonable heights for individual trees, and how many trees to add to the randomized forest – after a certain point the improvement achieved by adding more trees is negligible, and after a certain point building taller trees doesn't improve much the classification accuracy of the forest. The values of these parameters need to be tuned for the specific task being solved.



Decision tree forests are well suited to many classification tasks, and in terms of classification efficiency, are quite good – they have been incorporated into real-time classification systems you may have encountered before including the Kinect's body-pose recognition system that tracks users' movements in real time.

An interesting thought for you is – how does the performance of a randomized tree forest compare to that of a neural network? It's not a trivial question, but you should keep it in mind when you are faced with a classification problem in the future: What do you choose? There are implementation considerations, computational considerations, and performance considerations and you should have a solid idea of how the trees and the neural nets compare in each category so you can make a good choice.

***Alright, trees are cool but why do we call them AI?***

You can imagine how you may use a randomized tree forest for medical diagnosis – a feature vector for a patient can contain information from an interview, the results of any lab or imaging tests available, and previous medical history features. A randomized forest then provides a likely diagnosis based on these features.

It's a classification problem: We have a patient's feature vector and we're trying to determine which disease class (or no disease) it is more likely to belong to.

Many decision problems can be cast as classification problems, and planning tasks can also be approached with carefully built decision trees (should I bring an umbrella to work tomorrow? it's a classification problem with two possible classes, yes/no).

***That's that for this AI course! I hope it will be very useful to you in the future!***

Look back at what we have covered, think about the techniques and algorithms we implemented and how they may apply to future problems you may be faced with, and make sure you remember that AI at the end of the day is not so much about the algorithms and cool sounding systems (Deep Learning! Generative Adversarial Networks, etc.) but about:

- How to formulate a problem so we can approach it with one of our AI techniques
- How to define a good goal, utility function, heuristic, or loss function so that our AI technique can arrive at a good solution
- How to test and improve upon a solution we have, and how to decide if we have pushed a given technique as far as it will go and we need to think of something else
- Where each of the techniques you learned can fail, and how, so you can prepare for it and set up your system so as to avoid these pitfalls

Those are the important issues that require a trained AI specialist to solve – implementations of all the algorithms we covered already exist and are easily available in a variety of languages and platforms. So focus on the skills that will make you valuable: Taking a problem and figuring out how to formulate and set it up so it can be solved with any of the existing tools.

And if you want, go to grad school and push the boundary a bit further!