Introduction to Machine Learning (by Implementation)

Lecture 8: Decision Trees

Ian J. Watson

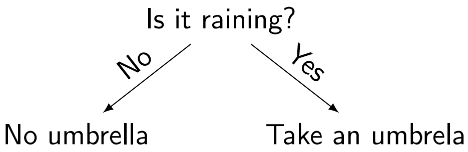
University of Seoul

University of Seoul Graduate Course 2019

Introduction

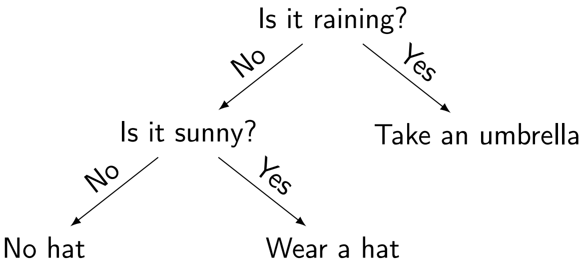
- We've spent several weeks building up the pieces of neural networks, today we'll change to a different direction
- We'll start the Decision Tree path
- As before, the setup is we have some input in $\mathbb{R}^n$ with known labels
- We want to find a function that will send the known inputs to the correct label and generalize to unseen data
  - Generalize = the procedure should correctly classify unseen input

Decision Trees



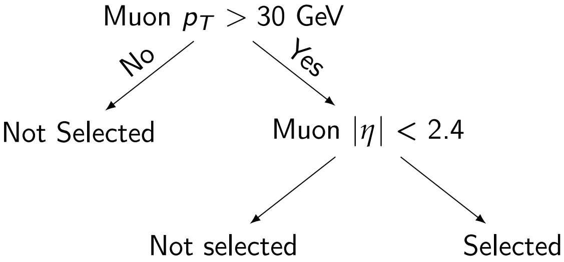
- Decision trees give a path to a result based on some conditions

Decision Trees



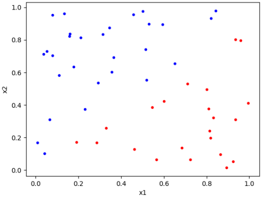
- Decision trees give a path to a result based on some conditions
- There could be several inputs, with multiple kinds of outputs
  - But always evaluate from top node down
- For true/false boolean inputs, straightforward to enumerate all options

Some Decision Tree Examples



- In the case of real valued inputs, we have to be more careful
- We can create left/right branches by asking for a value to be above/below some cut-off
  - We turn a real value variable into a binary decision at each node

Decision Trees with Real Numbers

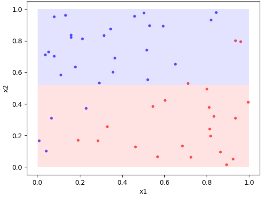


- Given a set of data we want to split into red and blue spaces

Decision Trees with Real Numbers

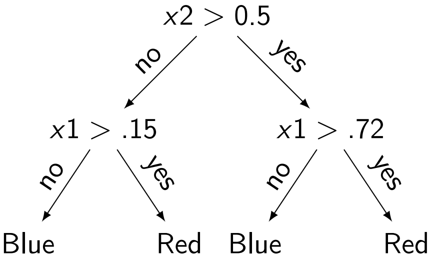
$x2 > 0.5$

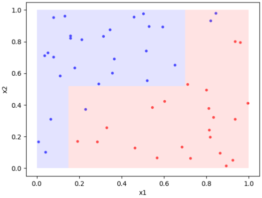
Red | Blue



- Given a set of data we want to split into red and blue spaces
- The decision tree will partition the problem space into discrete regions

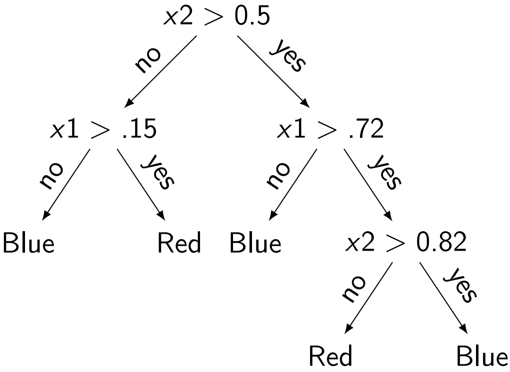
Decision Trees with Real Numbers

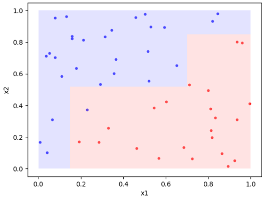




- Given a set of data we want to split into red and blue spaces
- The decision tree will partition the problem space into discrete regions
- Can add *levels* to split the space up further and further

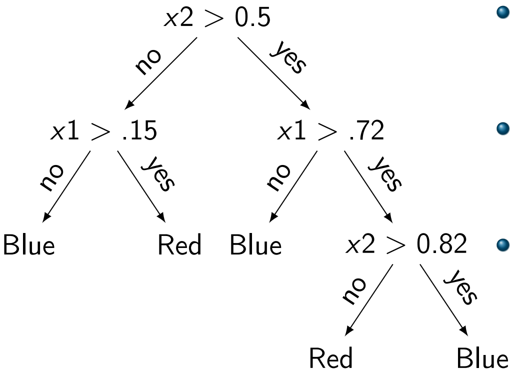
Decision Trees with Real Numbers





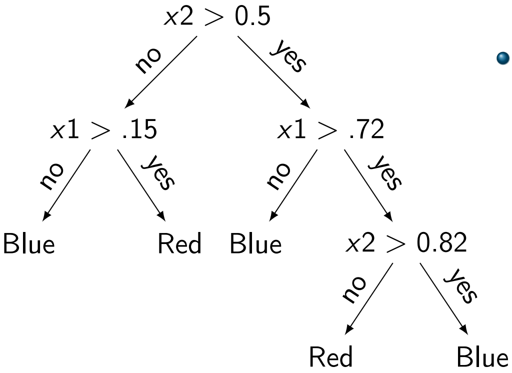
- Given a set of data we want to split into red and blue spaces
- The decision tree will partition the problem space into discrete regions
- Can add *levels* to split the space up further and further

Representation of a cut-offs in Python



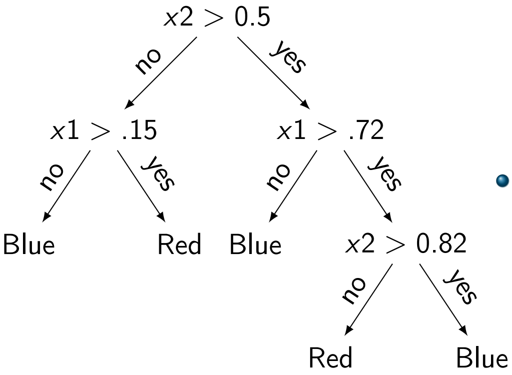
- Notice we can always write the cuts as $x_i > c$ for some $i \in \mathbb{Z}$ and $c \in \mathbb{R}$
- Our input will be a list of numbers, `x = [x0, x1, x2, x3, ...]`
- We will therefore only represent the i and c in our python code `(i,c)` will represent a node in the tree requiring $x_i > c$ at the node: `x[i] > c`

Tree Representation in Python



- Then, we need a way to store the tree structure
- A *node* on the tree can be:
  - *branch node*, or decision point, in which case we represent it as `[(i, c), left, right]` where `(i,c)` is the cutoff of the node, and left and right are subtrees representing the no and yes case respectively
  - *leaf node*, or an output, in which case we simply give the value that should be output

Tree Representation in Python



- Let the blue outputs be represented by 0 and red by 1

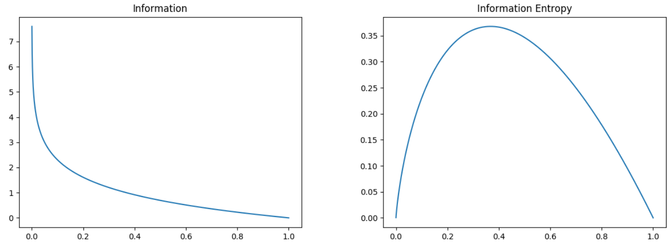
- This tree could be represented as:

```
[(2, 0.5),
 [(1, 0.15), 0, 1],
 [(1, 0.72), 0,
             [(2, 0.82), 1, 0]]]
```

Exercise

- Write `is_tree(thing)` which returns true only if:
  - thing is a list (test using `isinstance(thing, list)`) and the length is 3, and `thing[0]` is a tuple (test `isinstance(thing, tuple)`)
  - Remember the structure: `[(i, c), left, right]`
  - This is so we can have output lists as well as single numbers
- Write the function `classify(tree, data)` which takes a tree list and input list, and calculates the classification of the data based on the tree
- This will need to be written *recursively*
  - At a node:
    - Check if the node is a tree
    - If so, check the condition and *call classify with the correct subtree*
    - If not, then we're done, and you can output the value
- `tree_accuracy(x, y, tree)`
  - Given a list of data `x` and the corresponding correct outputs `y`, calculates the accuracy of the `tree` (correct / total) [using =classify=]

Shannon's Information Entropy



- Given a dataset with labels indexed by j, we define the information from observing label j as $I_j = -\log_2 p_j$
  - where p_j represents the probability of label j to be in the dataset (i.e. the fraction of data with label j)
    - If you put all the data in a hat and randomly picked one, what's the chance its in the j category
  - A low probability event "carries more information" than a high probability event (the theory was developed for communication)
- Then, we define entropy as $S = -\sum_j p_j \log_2 p_j$
  - The average information expected from sampling the data once
- As category prob. goes to 0 or 1, entropy goes to 0

Partition Entropy

- What's this to do with decision trees?
- Well, (next week) we will start off with the full dataset, then begin partitioning the data via our cutoffs
  - That is introduce branches to separate the data
- We need a measure of how much better we separate the categories after some new branch, we want to go high entropy to low entropy
  - But taking the entropy of the whole dataset always results in the same entropy
- Instead, we will test this by checking the *partition entropy*
- After partitioning the dataset Ω into subsets $\Omega_1, \Omega_2, \ldots$ (think, the data at each of the leaves of the tree), containing $q_1, q_2, q_3, \ldots$ fraction of the data
  - $S = q_1 S(\Omega_1) + q_2 S(\Omega_2) + \ldots$ is the partition entropy
  - i.e. the weighted average entropy of the subsets

Comments on partition entropy

- Good branch splits should
  - Put a large fraction of the data on either branch
  - Have each branch result in lower entropy (less random, more into individual classes)
- If you split one element of on left branch, put everything else on the right, then the left is very small entropy but doesn't help very much in the classification
- If you split the data 50-50 but each branch is equally random, this also hasn't helped
- Information is also called "surprisal", given a low-entropy set, you are "surprised" if you pick out a low probability label
  - Our goal is to minimize the "surpisal" of our splits

Exercises

- `entropy(class_probabilities)`
  - Takes a list of class probabilities and computes the entropy
- `class_probabilities(labels)`
  - Given a list of labels, returns a list of probabilities of labels
  - output list is unlabelled, only the probabilities are returned
- `data_entropy(labeled_data)`
  - `labeled_data` is in the from `(x, y)` (where `x` and `y` may be lists), return the entropy of the data based on the label `y`
- `partition_entropy(subsets)`
  - Given several subsets of the data ie a list of from `[subset1, subset2, ...]` where each subset is in the form of `labeled_data` above, return the partition entropy = weighted average of the entropy