CHAPTER 12

# Learning with Trees

We are now going to consider a rather different approach to machine learning, starting with one of the most common and powerful data structures in the whole of computer science: the binary tree. The computational cost of making the tree is fairly low, but the cost of using it is even lower: $\mathcal{O}(\log N)$, where $N$ is the number of datapoints. This is important for machine learning, since querying the trained algorithm should be as fast as possible since it happens more often, and the result is often wanted immediately. This is sufficient to make trees seem attractive for machine learning. However, they do have other benefits, such as the fact that they are easy to understand (following a tree to get a classification answer is transparent, which makes people trust it more than getting an answer from a 'black box' neural network).

For these reasons, classification by decision trees has grown in popularity over recent years. You are very likely to have been subjected to decision trees if you've ever phoned a helpline, for example for computer faults. The phone operators are guided through the decision tree by your answers to their questions.

The idea of a decision tree is that we break classification down into a set of choices about each feature in turn, starting at the root (base) of the tree and progressing down to the leaves, where we receive the classification decision. The trees are very easy to understand, and can even be turned into a set of if-then rules, suitable for use in a rule induction system.

In terms of optimisation and search, decision trees use a greedy heuristic to perform search, evaluating the possible options at the current stage of learning and making the one that seems optimal at that point. This works well a surprisingly large amount of the time.

## 12.1 USING DECISION TREES

As a student it can be difficult to decide what to do in the evening. There are four things that you actually quite enjoy doing, or have to do: going to the pub, watching TV, going to a party, or even (gasp) studying. The choice is sometimes made for you—if you have an assignment due the next day, then you need to study, if you are feeling lazy then the pub isn't for you, and if there isn't a party then you can't go to it. You are looking for a nice algorithm that will let you decide what to do each evening without having to think about it every night. Figure 12.1 provides just such an algorithm.

Each evening you start at the top (root) of the tree and check whether any of your friends know about a party that night. If there is one, then you need to go, regardless. Only if there is not a party do you worry about whether or not you have an assignment deadline coming up. If there is a crucial deadline, then you have to study, but if there is nothing that is urgent for the next few days, you think about how you feel. A sudden burst of energy

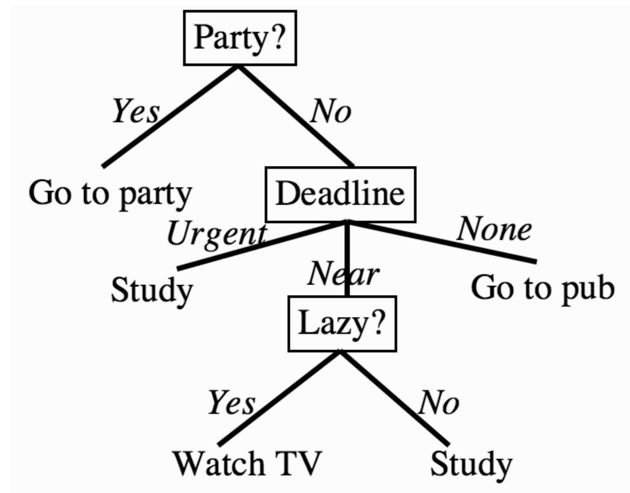


FIGURE 12.1 A simple decision tree to decide how you will spend the evening.

might make you study, but otherwise you'll be slumped in front of the TV indulging your secret love of Shortland Street (or other soap opera of your choice) rather than studying. Of course, near the start of the semester when there are no assignments to do, and you are feeling rich, you'll be in the pub.

One of the reasons that decision trees are popular is that we can turn them into a set of logical disjunctions (`if ... then` rules) that then go into program code very simply—the first part of the tree above can be turned into:

- `if` *there is a party* `then` *go to it*
- `if` *there is not a party* `and` *you have an urgent deadline* `then` *study*
- etc.

That's all that there is to using the decision tree. Compare it to the previous use of this data, with the Naïve Bayes Classifier in Section 2.3.2. The far more interesting part is how to construct the tree from data, and that is the focus of the next section.

## 12.2 CONSTRUCTING DECISION TREES

In the example above, the three features that we need for the algorithm are the state of your energy level, the date of your nearest deadline, and whether or not there is a party tonight. The question we need to ask is how, based on those features, we can construct the tree. There are a few different decision tree algorithms, but they are almost all variants of the same principle: the algorithms build the tree in a greedy manner starting at the root, choosing the most informative feature at each step. We are going to start by focusing on the most common: Quinlan's ID3, although we'll also mention its extension, known as C4.5, and another known as CART.

There was an important word hidden in the sentence above about how the trees work, which was informative. Choosing which feature to use next in the decision tree can be thought of as playing the game '20 Questions', where you try to elicit the item your opponent is thinking about by asking questions about it. At each stage, you choose a question that gives you the most information given what you know already. Thus, you would ask 'Is it an animal?' before you ask 'Is it a cat?'. The idea is to quantify this question of how much

information is provided to you by knowing certain facts. Encoding this mathematically is the task of information theory.

### 12.2.1 Quick Aside: Entropy in Information Theory

Information theory was 'born' in 1948 when Claude Shannon published a paper called "A Mathematical Theory of Communication." In that paper, he proposed the measure of information entropy, which describes the amount of impurity in a set of features. The entropy $H$ of a set of probabilities $p_i$ is (for those who know some physics, the relation to physical entropy should be clear):

$$\text{Entropy}(p) = -\sum_i p_i \log_2 p_i, \tag{12.1}$$

where the logarithm is base 2 because we are imagining that we encode everything using binary digits (bits), and we define $0 \log 0 = 0$. A graph of the entropy is given in Figure 12.2. Suppose that we have a set of positive and negative examples of some feature (where the feature can only take 2 values: positive and negative). If all of the examples are positive, then we don't get any extra information from knowing the value of the feature for any particular example, since whatever the value of the feature, the example will be positive. Thus, the entropy of that feature is 0. However, if the feature separates the examples into 50% positive and 50% negative, then the amount of entropy is at a maximum, and knowing about that feature is very useful to us. The basic concept is that it tells us how much *extra* information we would get from knowing the value of that feature. A function for computing the entropy is very simple, as here:

```
def calc_entropy(p):

    if p!=0:
        return -p * np.log2(p)
    else:
        return 0
```

For our decision tree, the best feature to pick as the one to classify on now is the one that gives you the most information, i.e., the one with the highest entropy. After using that feature, we re-evaluate the entropy of each feature and again pick the one with the highest entropy.

Information theory is a very interesting subject. It is possible to download Shannon's 1948 paper from the Internet, and also to find many resources showing where it has been applied. There are now whole journals devoted to information theory because it is relevant to so many areas such as computer and telecommunication networks, machine learning, and data storage. Some further readings in the area are given at the end of the chapter.

### 12.2.2 ID3

Now that we have a suitable measure for choosing which feature to choose next, entropy, we just have to work out how to apply it. The important idea is to work out how much the entropy of the whole training set would decrease if we choose each particular feature for the next classification step. This is known as the information gain, and it is defined as

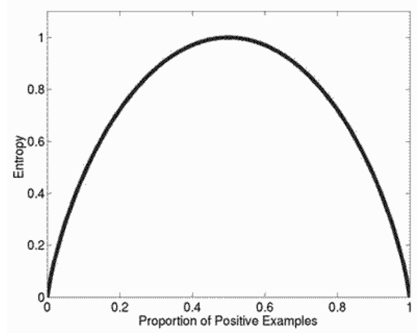


FIGURE 12.2 A graph of entropy, detailing how much information is available from finding out another piece of information given what you already know.

the entropy of the whole set minus the entropy when a particular feature is chosen. This is defined by (where S is the set of examples, F is a possible feature out of the set of all possible ones, and $|S_f|$ is a count of the number of members of $S$ that have value $f$ for feature $F$):

$$\text{Gain}(S, F) = \text{Entropy}(S) - \sum_{f \in values(F)} \frac{|S_f|}{|S|} \text{Entropy}(S_f). \tag{12.2}$$

As an example, suppose that we have data (with outcomes) $S = \{s_1 = \text{true}, s_2 = \text{false}, s_3 = \text{false}, s_4 = \text{false}\}$ and one feature $F$ that can have values $\{f_1, f_2, f_3\}$. In the example, the feature value for $s_1$ could be $f_2$, for $s_2$ it could be $f_2$, for $s_3$, $f_3$ and for $s_4$, $f_1$ then we can calculate the entropy of $S$ as (where $\oplus$ means true, of which we have one example, and $\ominus$ means false, of which we have three examples):

$$\begin{aligned} \text{Entropy}(S) &= -p_\oplus \log_2 p_\oplus - p_\ominus \log_2 p_\ominus \\ &= -\frac{1}{4} \log_2 \frac{1}{4} - \frac{3}{4} \log_2 \frac{3}{4} \\ &= 0.5 + 0.311 = 0.811. \end{aligned} \tag{12.3}$$

The function Entropy$(S_f)$ is similar, but only computed with the subset of data where feature $F$ has values $f$.

If you were trying to follow those calculations on a calculator, you might be wondering how to compute $\log_2 p$. The answer is to use the identity $\log_2 p = \ln p / \ln(2)$, where ln is the natural logarithm, which your calculator can produce. NumPy has the `np.log2()` function.

We now want to compute the information gain of $F$, so we now need to compute each of the values inside the summation in Equation (12.2), $\frac{|S_f|}{|S|}\text{Entropy}(S)$ (in our example, the features are 'Deadline', 'Party', and 'Lazy'):

$$\begin{aligned}
\frac{|S_{f_1}|}{|S|}\text{Entropy}(S_{f_1}) &= \frac{1}{4} \times \left(-\frac{0}{1}\log_2\frac{0}{1} - \frac{1}{1}\log_2\frac{1}{1}\right) \\
&= 0 \qquad (12.4) \\
\frac{|S_{f_2}|}{|S|}\text{Entropy}(S_{f_2}) &= \frac{2}{4} \times \left(-\frac{1}{2}\log_2\frac{1}{2} - \frac{1}{2}\log_2\frac{1}{2}\right) \\
&= \frac{1}{2} \qquad (12.5) \\
\frac{|S_{f_3}|}{|S|}\text{Entropy}(S_{f_3}) &= \frac{1}{4} \times \left(-\frac{0}{1}\log_2\frac{0}{1} - \frac{1}{1}\log_2\frac{1}{1}\right) \\
&= 0 \qquad (12.6)
\end{aligned}$$

The information gain from adding this feature is the entropy of S minus the sum of the three values above:

$$\text{Gain}(S, F) = 0.811 - (0 + 0.5 + 0) = 0.311. \qquad (12.7)$$

This can be computed in an algorithm using the following function (where lots of the code is to get the relevant data):

```
def calc_info_gain(data,classes,feature):
    gain = 0
    nData = len(data)
    # List the values that feature can take
    values = []
    for datapoint in data:
        if datapoint[feature] not in values:
            values.append(datapoint[feature])

    featureCounts = np.zeros(len(values))
    entropy = np.zeros(len(values))
    valueIndex = 0
    # Find where those values appear in data[feature] and the corresponding 
    class
    for value in values:
        dataIndex = 0
        newClasses = []
        for datapoint in data:
            if datapoint[feature]==value:
                featureCounts[valueIndex]+=1
                newClasses.append(classes[dataIndex])
            dataIndex += 1

        # Get the values in newClasses
        classValues = []
        for aclass in newClasses:
            if classValues.count(aclass)==0:
                classValues.append(aclass)
```

```
        classCounts = np.zeros(len(classValues))
        classIndex = 0
        for classValue in classValues:
            for aclass in newClasses:
                if aclass == classValue:
                    classCounts[classIndex]+=1
            classIndex += 1

        for classIndex in range(len(classValues)):
            entropy[valueIndex] += calc_entropy(float(classCounts[classIndex])
            /sum(classCounts))
        gain += float(featureCounts[valueIndex])/nData * entropy[valueIndex]
        valueIndex += 1
    return gain
```

The ID3 algorithm computes this information gain for each feature and chooses the one that produces the highest value. In essence, that is all there is to the algorithm. It searches the space of possible trees in a greedy way by choosing the feature with the highest information gain at each stage. The output of the algorithm is the tree, i.e., a list of nodes, edges, and leaves. As with any tree in computer science, it can be constructed recursively. At each stage the best feature is selected and then removed from the dataset, and the algorithm is recursively called on the rest. The recursion stops when either there is only one class remaining in the data (in which case a leaf is added with that class as its label), or there are no features left, when the most common label in the remaining data is used.

**The ID3 Algorithm**

- If all examples have the same label:
  - return a leaf with that label
- Else if there are no features left to test:
  - return a leaf with the most common label
- Else:
  - choose the feature $\hat{F}$ that maximises the information gain of $S$ to be the next node using Equation (12.2)
  - add a branch from the node for each possible value $f$ in $\hat{F}$
  - for each branch:
    - calculate $S_f$ by removing $\hat{F}$ from the set of features
    - recursively call the algorithm with $S_f$, to compute the gain relative to the current set of examples

Owing to the focus on classification for real-world examples, trees are often used with text features rather than numeric values. This makes it rather difficult to use NumPy, and so the sample implementation is pretty well pure Python. It uses a feature of Python that is uncommon in other languages, which is the dictionary in order to hold the tree, which uses the braces $\{,\}$, and which is described next before we look at the decision tree implementation.

### 12.2.3 Implementing Trees and Graphs in Python

Trees are really just a restricted version of graphs, since they both consist of nodes and edges between the nodes. Graphs are a very useful data structure in many different areas of computer science. There are two reasonable ways to represent a graph computationally. One is as an $N \times N$ matrix, where $N$ is the number of nodes in the network. Each element of the matrix is a 1 if there is a link between the two nodes, and a 0 otherwise. The benefit of this approach is that it is easy to give weights to the links by changing the 1s to the values of the weights. The alternative is to store a list of nodes, following each by a list of nodes that it is linked to. Both are fairly natural in Python, with the second making use of the dictionary, a basic data structure that we have not used much, except for very simply in the decision tree (Chapter 12) that consists of a set of keys and values. For a graph, the key to each dictionary entry is the name of the node, and its value is a list of the nodes that it is connected to, as in this example:

```
graph = {'A': ['B', 'C'],'B': ['C', 'D'],'C': ['D'],'D': ['C'],'E': ['F'],
'F': ['C']}
```

That is all there is to it for creating the dictionary, and using it is not very different, since there are built-in methods to get a list of keys (`keys()`) and check if a key is in a dictionary (`in`). Code to find a path through the graph can then be written as a simple recursive function:

```
def findPath(graph, start, end, pathSoFar):
    pathSoFar = pathSoFar + [start]
    if start == end:
        return pathSoFar
    if start not in graph:
        return None
    for node in graph[start]:
        if node not in pathSoFar:
            newpath = findPath(graph, node, end, pathSoFar)
            return newpath
    return None
```

Using those methods we can now look at a Python implementation of the decision tree, which also has a recursive function call as its basis.

### 12.2.4 Implementation of the Decision Tree

The `make_tree()` function (which uses the `calc_entropy()` and `calc_info_gain()` functions that were described previously) looks like:

```
def make_tree(data,classes,featureNames):
    # Various initialisations suppressed
```

```
default = classes[np.argmax(frequency)]
if nData==0 or nFeatures == 0:
    # Have reached an empty branch
    return default
elif classes.count(classes[0]) == nData:
    # Only 1 class remains
    return classes[0]
else:
    # Choose which feature is best
    gain = np.zeros(nFeatures)
    for feature in range(nFeatures):
        g = calc_info_gain(data,classes,feature)
        gain[feature] = totalEntropy - g
    bestFeature = np.argmax(gain)
    tree = {featureNames[bestFeature]:{}}
    # Find the possible feature values
    for value in values:
        # Find the datapoints with each feature value
        for datapoint in data:
            if datapoint[bestFeature]==value:
                if bestFeature==0:
                    datapoint = datapoint[1:]
                    newNames = featureNames[1:]
                elif bestFeature==nFeatures:
                    datapoint = datapoint[:-1]
                    newNames = featureNames[:-1]
                else:
                    datapoint = datapoint[:bestFeature]
                    datapoint.extend(datapoint[bestFeature+1:])
                    newNames = featureNames[:bestFeature]
                    newNames.extend(featureNames[bestFeature+1:])
                newData.append(datapoint)
                newClasses.append(classes[index])
            index += 1
        # Now recurse to the next level
        subtree = make_tree(newData,newClasses,newNames)
        # And on returning, add the subtree on to the tree
        tree[featureNames[bestFeature]][value] = subtree
    return tree
```

It is worth considering how ID3 generalises from training examples to the set of all possible inputs. It uses a method known as the inductive bias. The choice of the next feature to add into the tree is the one with the highest information gain, which biases the algorithm towards smaller trees, since it tries to minimise the amount of information that is left. This is consistent with a well-known principle that short solutions are usually better than longer ones (not necessarily true, but simpler explanations are usually easier to remember and understand). You might have heard of this principle as 'Occam's Razor', although I prefer

it as an acronym: KISS (Keep It Simple, Stupid). In fact, there is a sound information-theoretic way to write down this principle. It is known as the Minimum Description Length (MDL) and was proposed by Rissanen in 1989. In essence it says that the shortest description of something, i.e., the most compressed one, is the best description.

Note that the algorithm can deal with noise in the dataset, because the labels are assigned to the most common value of the target attribute. Another benefit of decision trees is that they can deal with missing data. Think what would happen if an example has a missing feature. In that case, we can skip that node of the tree and carry on without it, summing over all the possible values that that feature could have taken. This is virtually impossible to do with neural networks: how do you represent missing data when the computation is based on whether or not a neuron is firing? In the case of neural networks it is common to either throw away any datapoints that have missing data, or guess (more technically impute any missing values, either by identifying similar datapoints and using their value or by using the mean or median of the data values for that feature). This assumes that the data that is missing is randomly distributed within the dataset, not missing because of some unknown process.

Saying that ID3 is biased towards short trees is only partly true. The algorithm uses all of the features that are given to it, even if some of them are not necessary. This obviously runs the risk of overfitting, indeed it makes it very likely. There are a few things that you can do to avoid overfitting, the simplest one being to limit the size of the tree. You can also use a variant of early stopping by using a validation set and measuring the performance of the tree so far against it. However, the approach that is used in more advanced algorithms (most notably C4.5, which Quinlan invented to improve on ID3) is pruning.

There are a few versions of pruning, all of which are based on computing the full tree and reducing it, evaluating the error on a validation set. The most naïve version runs the decision tree algorithm until all of the features are used, so that it is probably overfitted, and then produces smaller trees by running over the tree, picking each node in turn, and replacing the subtree beneath every node with a leaf labelled with the most common classification of the subtree. The error of the pruned tree is evaluated on the validation set, and the pruned tree is kept if the error is the same as or less than the original tree, and rejected otherwise.

C4.5 uses a different method called rule post-pruning. This consists of taking the tree generated by ID3, converting it to a set of if-then rules, and then pruning each rule by removing preconditions if the accuracy of the rule increases without it. The rules are then sorted according to their accuracy on the training set and applied in order. The advantages of dealing with rules are that they are easier to read and their order in the tree does not matter, just their accuracy in the classification.

### 12.2.5 Dealing with Continuous Variables

One thing that we have not yet discussed is how to deal with continuous variables, we have only considered those with discrete sets of feature values. The simplest solution is to discretise the continuous variable. However, it is also possible to leave it continuous and modify the algorithm. For a continuous variable there is not just one place to split it: the variable can be broken between any pair of datapoints, as shown in Figure 12.3. It can, of course, be split in any of the infinite locations along the line as well, but they are no different to this smaller set of locations. Even this smaller set makes the algorithm more expensive for continuous variables than it is for discrete ones, since as well as calculating the information gain of each variable to pick the best one, the information gain of many points within each variable has to be computed. In general, only one split is made to a continuous

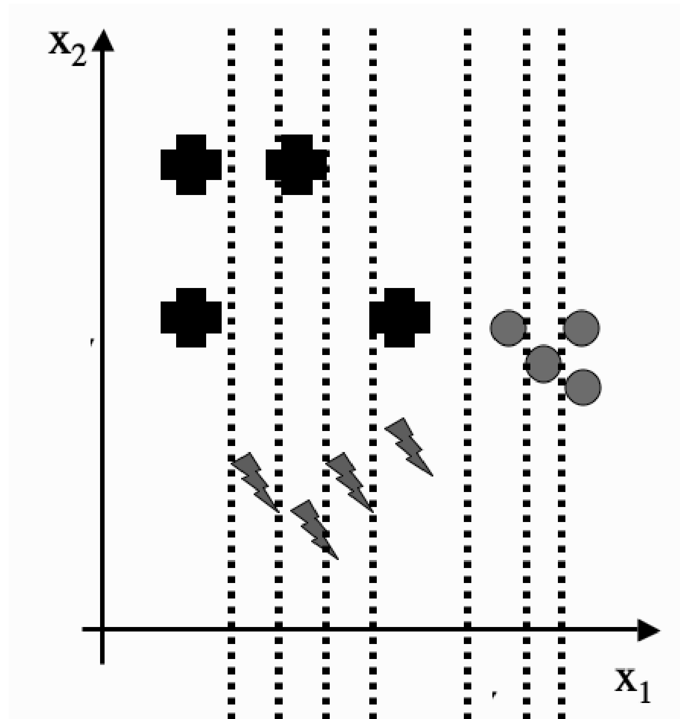


FIGURE 12.3 Possible places to split the variable $x_1$, between each of the datapoints as the feature value increases.

variable, rather than allowing for threeway or higher splits, although these can be done if necessary.

The trees that these algorithms make are all univariate trees, because they pick one feature (dimension) at a time and split according to that one. There are also algorithms that make multivariate trees by picking combinations of features. This can make for considerably smaller trees if it is possible to find straight lines that separate the data well, but are not parallel to any axis. However, univariate trees are simpler and tend to get good results, so we won't consider multivariate trees any further. This fact that one feature is chosen at a time provides another useful way to visualise what the decision tree is doing. Figure 12.4 shows the idea. Given a dataset that contains three classes, the algorithm picks a feature and value for that feature to split the remaining data into two. The final tree that results from this is shown in Figure 12.5.

### 12.2.6 Computational Complexity

The computational cost of constructing binary trees is well known for the general case, being $\mathcal{O}(N \log N)$ for construction and $\mathcal{O}(\log N)$ for returning a particular leaf, where $N$ is the number of nodes. However, these results are for balanced binary trees, and decision trees are often not balanced; while the information measures attempt to keep the tree balanced by finding splits that separate the data into two even parts (since that will have the largest entropy), there is no guarantee of this. Nor are they necessarily binary, especially for ID3 and C4.5, as our example shows.

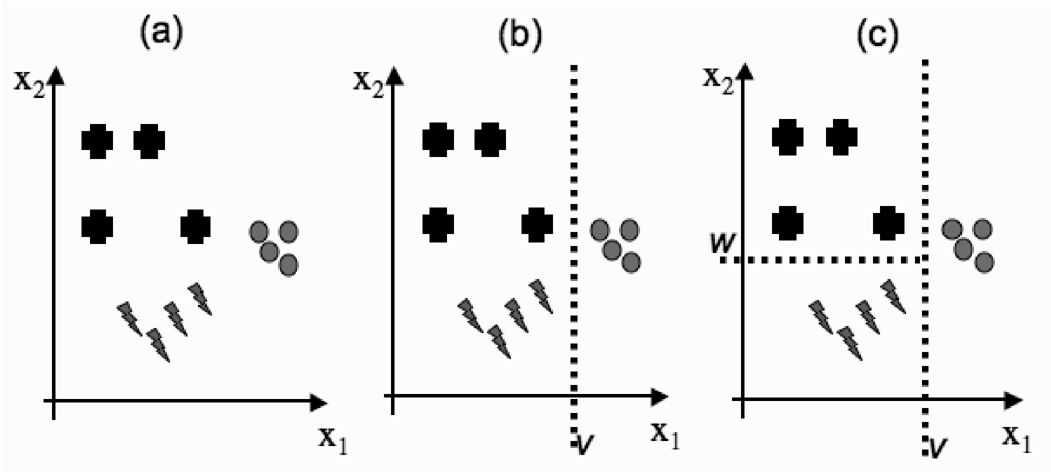


FIGURE 12.4 The effect of decision tree choices. The two-dimensional dataset shown in (a) is split first by choosing feature $x_1$ (b) and then $x_2$, (c) which separates out the three classes. The final tree is shown in Figure 12.5.

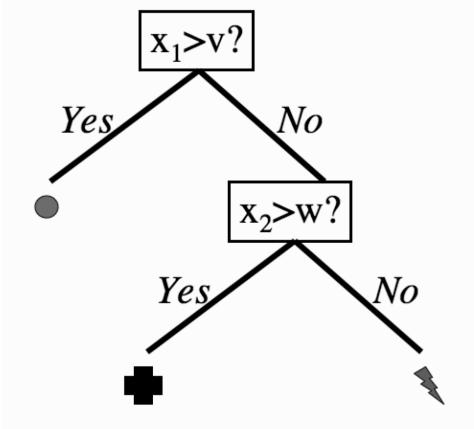


FIGURE 12.5 The final tree created by the splits in Figure 12.4.

If we assume that the tree is approximately balanced, then the cost at each node consists of searching through the $d$ possible features (although this decreases by 1 at each level, that doesn't affect the complexity in the $\mathcal{O}(\cdot)$ notation) and then computing the information gain for the dataset for each split. This has cost $\mathcal{O}(dn \log n)$, where $n$ is the size of the dataset at that node. For the root, $n = N$, and if the tree is balanced, then $n$ is divided by 2 at each stage down the tree. Summing this over the approximately $\log N$ levels in the tree gives computational cost $\mathcal{O}(dN^2 \log N)$.

## 12.3 CLASSIFICATION AND REGRESSION TREES (CART)

There is another well-known tree-based algorithm, CART, whose name indicates that it can be used for both classification and regression. Classification is not wildly different in CART, although it is usually constrained to construct binary trees. This might seem odd at first, but there are sound computer science reasons why binary trees are good, as suggested in the computational cost discussion above, and it is not a real limation. Even in the example that we started the chapter with, we can always turn questions into binary decisions by splitting the question up a little. Thus, a question that has three answers (say the question about when your nearest assignment deadline is, which is either 'urgent', 'near', or 'none') can be split into two questions: first, 'is the deadline urgent?', and then if the answer to that is 'no', second 'is the deadline near?' The only real difference with classification in CART is that a different information measure is commonly used. This is discussed next, before we look briefly at regression with trees.

### 12.3.1 Gini Impurity

The entropy that was used in ID3 as the information measure is not the only way to pick features. Another possibility is something known as the Gini impurity. The 'impurity' in the name suggests that the aim of the decision tree is to have each leaf node represent a set of datapoints that are in the same class, so that there are no mismatches. This is known as purity. If a leaf is pure then all of the training data within it have just one class. In which case, if we count the number of datapoints at the node (or better, the fraction of the number of datapoints) that belong to a class $i$ (call it $N(i)$), then it should be 0 for all except one value of $i$. So suppose that you want to decide on which feature to choose for a split. The algorithm loops over the different features and checks how many points belong to each class. If the node is pure, then $N(i) = 0$ for all values of $i$ except one particular one. So for any particular feature $k$ you can compute:

$$G_k = \sum_{i=1}^{c} \sum_{j \neq i} N(i)N(j), \tag{12.8}$$

where $c$ is the number of classes. In fact, you can reduce the algorithmic effort required by noticing that $\sum_i N(i) = 1$ (since there has to be some output class) and so $\sum_{j \neq i} N(j) = 1 - N(i)$. Then Equation (12.8) is equivalent to:

$$G_k = 1 - \sum_{i=1}^{c} N(i)^2. \tag{12.9}$$

Either way, the Gini impurity is equivalent to computing the expected error rate if the classification was picked according to the class distribution. The information gain can then be measured in the same way, subtracting each value $G_i$ from the total Gini impurity.

The information measure can be changed in another way, which is to add a weight to the misclassifications. The idea is to consider the cost of misclassifying an instance of class $i$ as class $j$ (which we will call the risk in Section 2.3.1) and add a weight that says how important each datapoint is. It is typically labelled as $\lambda_{ij}$ and is presented as a matrix, with element $\lambda_{ij}$ representing the cost of misclassifying $i$ as $j$. Using it is simple, modifying the Gini impurity (Equation (12.8)) to be:

$$G_i = \sum_{j \neq i} \lambda_{ij} N(i) N(j). \tag{12.10}$$

We will see in Section 13.1 that there is another benefit to using these weights, which is to successively improve the classification ability by putting higher weight on datapoints that the algorithm is getting wrong.

### 12.3.2 Regression in Trees

The new part about CART is its application in regression. While it might seem strange to use trees for regression, it turns out to require only a simple modification to the algorithm. Suppose that the outputs are continuous, so that a regression model is appropriate. None of the node impurity measures that we have considered so far will work. Instead, we'll go back to our old favourite—the sum-of-squares error. To evaluate the choice of which feature to use next, we also need to find the value at which to split the dataset according to that feature. Remember that the output is a value at each leaf. In general, this is just a constant value for the output, computed as the mean average of all the datapoints that are situated in that leaf. This is the optimal choice in order to minimise the sum-of-squares error, but it also means that we can choose the split point quickly for a given feature, by choosing it to minimise the sum-of-squares error. We can then pick the feature that has the split point that provides the best sum-of-squares error, and continue to use the algorithm as for classification.

## 12.4 CLASSIFICATION EXAMPLE

We'll work through an example using ID3 in this section. The data that we'll use will be a continuation of the one we started the chapter with, about what to do in the evening.

When we want to construct the decision tree to decide what to do in the evening, we start by listing everything that we've done for the past few days to get a suitable dataset (here, the last ten days):

| Deadline? | Is there a party? | Lazy? | Activity |
|---|---|---|---|
| Urgent | Yes | Yes | Party |
| Urgent | No | Yes | Study |
| Near | Yes | Yes | Party |
| None | Yes | No | Party |
| None | No | Yes | Pub |
| None | Yes | No | Party |
| Near | No | No | Study |
| Near | No | Yes | TV |
| Near | Yes | Yes | Party |
| Urgent | No | No | Study |

To produce a decision tree for this problem, the first thing that we need to do is work out which feature to use as the root node. We start by computing the entropy of $S$:

$$\begin{aligned}\text{Entropy}(S) &= -p_{\text{party}} \log_2 p_{\text{party}} - p_{\text{study}} \log_2 p_{\text{study}} \\ &- p_{\text{pub}} \log_2 p_{\text{pub}} - p_{\text{TV}} \log_2 p_{\text{TV}} \\ &= -\frac{5}{10}\log_2\frac{5}{10} - \frac{3}{10}\log_2\frac{3}{10} - \frac{1}{10}\log_2\frac{1}{10} - \frac{1}{10}\log_2\frac{1}{10} \\ &= 0.5 + 0.5211 + 0.3322 + 0.3322 = 1.6855 \qquad (12.11)\end{aligned}$$

and then find which feature has the maximal information gain:

$$\begin{aligned}\text{Gain}(S, \text{Deadline}) &= 1.6855 - \frac{|S_{\text{urgent}}|}{10}\text{Entropy}(S_{\text{urgent}}) \\ &- \frac{|S_{\text{near}}|}{10}\text{Entropy}(S_{\text{near}}) - \frac{|S_{\text{none}}|}{10}\text{Entropy}(S_{\text{none}}) \\ &= 1.6855 - \frac{3}{10}\left(-\frac{2}{3}\log_2\frac{2}{3} - \frac{1}{3}\log_2\frac{1}{3}\right) \\ &- \frac{4}{10}\left(-\frac{2}{4}\log_2\frac{2}{4} - \frac{1}{4}\log_2\frac{1}{4} - \frac{1}{4}\log_2\frac{1}{4}\right) \\ &- \frac{3}{10}\left(-\frac{1}{3}\log_2\frac{1}{3} - \frac{2}{3}\log_2\frac{2}{3}\right) \\ &= 1.6855 - 0.2755 - 0.6 - 0.2755 \\ &= 0.5345 \qquad (12.12)\end{aligned}$$

$$\begin{aligned}\text{Gain}(S, \text{Party}) &= 1.6855 - \frac{5}{10}\left(-\frac{5}{5}\log_2\frac{5}{5}\right) \\ &- \frac{5}{10}\left(-\frac{3}{5}\log_2\frac{3}{5} - \frac{1}{5}\log_2\frac{1}{5} - \frac{1}{5}\log_2\frac{1}{5}\right) \\ &= 1.6855 - 0 - 0.6855 \\ &= 1.0 \qquad (12.13)\end{aligned}$$

$$\begin{aligned}\text{Gain}(S, \text{Lazy}) &= 1.6855 - \frac{6}{10}\left(-\frac{3}{6}\log_2\frac{3}{6} - \frac{1}{6}\log_2\frac{1}{6} - \frac{1}{6}\log_2\frac{1}{6} - \frac{1}{6}\log_2\frac{1}{6}\right) \\ &- \frac{4}{10}\left(-\frac{2}{4}\log_2\frac{2}{4} - \frac{2}{4}\log_2\frac{2}{4}\right) \\ &= 1.6855 - 1.0755 - 0.4 \\ &= 0.21 \qquad (12.14)\end{aligned}$$

Therefore, the root node will be the party feature, which has two feature values ('yes' and 'no'), so it will have two branches coming out of it (see Figure 12.6). When we look at the 'yes' branch, we see that in all five cases where there was a party we went to it, so we just put a leaf node there, saying 'party'. For the 'no' branch, out of the five cases there are three different outcomes, so now we need to choose another feature. The five cases we are looking at are:

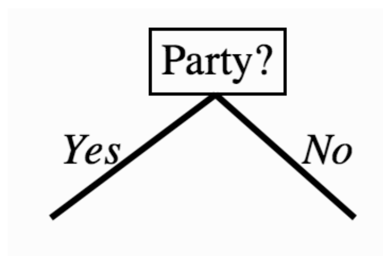


FIGURE 12.6 The decision tree after one step of the algorithm.

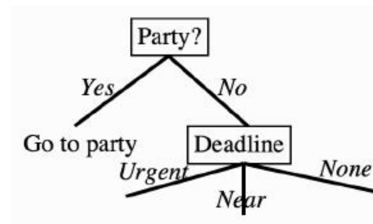


FIGURE 12.7 The tree after another step.

| Deadline? | Is there a party? | Lazy? | Activity |
|---|---|---|---|
| Urgent | No | Yes | Study |
| None | No | Yes | Pub |
| Near | No | No | Study |
| Near | No | Yes | TV |
| Urgent | No | Yes | Study |

We've used the party feature, so we just need to calculate the information gain of the other two over these five examples:

$$\begin{aligned}
\text{Gain}(S, \text{Deadline}) &= 1.371 - \frac{2}{5}\left(-\frac{2}{2}\log_2\frac{2}{2}\right) \\
&- \frac{2}{5}\left(-\frac{1}{2}\log_2\frac{1}{2} - \frac{1}{2}\log_2\frac{1}{2}\right) - \frac{1}{5}\left(-\frac{1}{1}\log_2\frac{1}{1}\right) \\
&= 1.371 - 0 - 0.4 - 0 \\
&= 0.971 \qquad (12.15) \\
\text{Gain}(S, \text{Lazy}) &= 1.371 - \frac{4}{5}\left(-\frac{2}{4}\log_2\frac{2}{4} - \frac{1}{4}\log_2\frac{1}{4} - \frac{1}{4}\log_2\frac{1}{4}\right) \\
&- \frac{1}{5}\left(-\frac{1}{1}\log_2\frac{1}{1}\right) \\
&= 1.371 - 1.2 - 0 \\
&= 0.1710 \qquad (12.16)
\end{aligned}$$

This leads to the tree shown in Figure 12.7. From this point it is relatively simple to complete the tree, leading to the one that was shown in Figure 12.1.

## FURTHER READING

For more information about decision trees, the following two books are of interest:


- J.R. Quinlan. *C4.5: Programs for Machine Learning.* Morgan Kaufmann, San Francisco, CA, USA, 1993.
- L. Breiman, J.H. Friedman, R.A. Olshen, and C.J. Stone. *Classification and Regression Trees.* Chapman & Hall, New York, USA, 1993.