3 Decision Trees

Ever play the game 20 questions? That's what a decision tree is — something which asks questions about an item until it can determine what label an item should have. Decision trees are the model produced by decision tree algorithms, and decision tree algorithms are classification algorithms. Decision trees divvy up the space of features into regions, each assigned a label.

Decision trees do this by assigning to each of their nonleaf nodes a **query** about some feature. Based on the answer to that query, we recurse down one or another subtree in the decision tree, recursively asking more query questions, until we come to a leaf node, which is labeled with a label: the decision tree's guess. Let's begin with decision trees designed for spaces where all the features are **categorical**, that is, each dimension of a feature whose possible values come from an unordered, finite set. We'll call such features **attributes**.[40]

By asking a question, the decision tree divides the world up into subsets, one for each possible answer to the question. If a decision tree was asking questions about animals, and the first question it asked was their genus, then the world is now divided into subsets like mammal, bird, amphibian, fish, reptile, etc. If a decision tree found out that the item it cared about was a fish, it might go on to ask another question — perhaps whether or not the animal was a live-bearer. But it wouldn't ask this question if the animal was a mammal, because pretty much all mammals are live-bearers. So the useful secondary questions are often different depending on what primary question you ask first.

Consider the following table, from p. 196 of "Artificial Intelligence: Theory and Practice":

ID	Attributes				Class Label
Room#	Status	Floor	Department	Size	Recycling Bin?
307	faculty	three	ee	large	no
309	staff	three	ee	small	no
408	faculty	four	cs	medium	yes
415	student	four	ee	large	yes
509	staff	five	cs	medium	no
517	faculty	five	cs	large	yes
316	student	three	ee	small	yes
420	staff	four	cs	medium	no

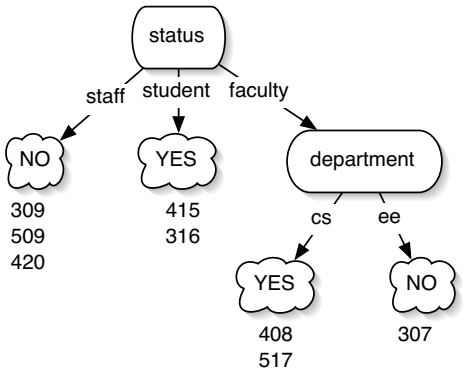
This is table of rooms by room number, plus some room facts. For each room, there are four attributes (status, floor, department, and size). There is also a label provided (whether the room has a recycling bin).

Each attribute can take on some number of *values*. For example, the "status" attribute can take on three values: "faculty", "staff", and "student". Likewise, the label can take on values. In this case there are two values for the label: "yes" and "no".

A decision tree algorithm asks questions about the room attributes and comes up with a decision which says what label the room is assigned. In a decision tree, the questions asked are the nonleaf nodes in the tree, which lead to new nonleaf nodes to ask depending on the answer to the attribute

[40] This is to distinguish them from the other side of the equation: the class label, which is of course also chosen from an unordered finite set. Another term for attributes is **categories**

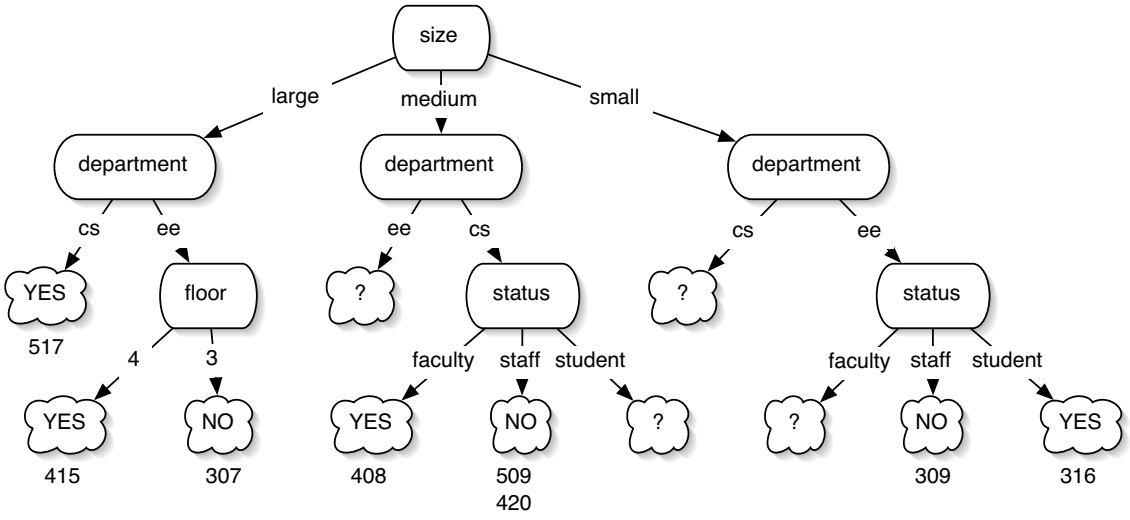
question. The leaf nodes decide what the label should be. Here's an example decision tree (also from "Artificial Intelligence: Theory and Practice"):



In this decision tree, after asking status, *none* the rooms that are staff rooms have recycling bins. And *all* the rooms that are student rooms have recycling bins. So if the room is a student room, we know its classification immediately, and likewise for staff rooms. If the room is a faculty room, we have to ask a further question. The department question nicely divides up the recycling bins from the non-recycling bins.

How many questions must be asked on average before we know the label of the room? There are 8 rooms. 5 of these rooms only need 1 question. 3 rooms need 2 questions. So the answer is 1(5/8) + 2(3/8) = 1.375 questions on average.

But it depends on how you order your questions. By asking some *bad* questions first (asking size, then department, then either floor or status), you wind up having to ask 3 questions on average; furthermore, many of the questions lead to answers for which we don't have any examples — so we don't know how to classify them, and will have to make a guess. Here's the resulting tree, again from "Artificial Intelligence: Theory and Practice":



Yuck! What a mess of a decision tree. Why do we want small decision trees? Because large decision trees tend to overfit the data: thus it is generally believed that small decision trees tend to

do a better job of generalization. Our compact decision tree was able to lump a bunch of rooms together under the same leaf node. In the large decision tree, pretty much every room wound up with its own set of questions. The theory goes that the big decision tree basically came up with special-purpose questions (thus overfitting) for each separate room, rather than finding general traits that rooms share. We want to find general traits.

3.1 Queryting a Decision Tree

Given the diagrams above, you can probably guess how to query a decision tree once it's been generated. Every child of every node in the tree is associated with a **query** which may or may not be true about some sample. We recurse down the branches for which the queries are true until we come to a leaf node:

Algorithm 4 *QueryDecisionTree*

1: $T \leftarrow$ decision tree with root r
2: $\vec{x}' \leftarrow$ as of-yet unseen sample to query, of dimensionality d

3: **if** r is a leaf node **then**
4: **return** the label assigned to r
5: **else**
6: $child \leftarrow$ the child subtree of r whose query is true for $\vec{x}'$
7: **return** QueryDecisionTree($child$, $\vec{x}'$)

There will be exactly one child subtree of a node for which the query is true: you don't have to worry about that (no overspecification or underspecification).

3.2 Building a Decision Tree

(This assumes our features are all categorical. We'll get to non-categorical features in a moment.)

Building a decision tree is also recursive. We take an existing set of samples, then choose a feature to ask a question about. We create one subtree for each possible value that f could take on, then recurse to build those subtrees. For each subtree, the only samples of interest are the ones for whom f had the value assigned to that subtree. Once we've asked a question about f in a given subtree, it's no longer interesting, so we remove it from consideration in subsubtrees.[41]

Since we're recursing, we need some base cases to know when to stop. There are three:

- Do all of our samples have the same label? If so it's easy: we're just a leaf node labelled with that label.
- Do we have any samples left? If not, this happens when the question asked in our parent resulted in absolutely no samples coming to us (not a good situation, but sometimes it happens). We can't recurse any more, and must create a leaf node with a label. We have to pick *some* label to use: we'll use a **default label** l_*. Usually l_* is defined as the most

[41] Otherwise we'd have 20-question games like: "Are you bigger than a bread box?"' ... "yes" ... "Okay. So then, are you bigger than a breadbox?"

common label among the samples that were used to generate our *parent* node.[42] In order to determine the most common label, we'll need a little function called MostCommonLabel(X): this simply returns the label which appears the most often among the samples in X, breaking ties arbitrarily.[43]

- Do we have any features left with which to ask questions, yet still have samples with different labels? If not, then we have some samples with identical features but different labels, and there's nothing we can do about it. Pick a label, usually the most common label among the remaining samples.

Armed with this, perhaps the following algorithm will now make sense:

Algorithm 5 *BuildDecisionTree*

$X \leftarrow \{\langle \vec{x}^{(1)}, l_1 \rangle, ..., \langle \vec{x}^{(n)}, l_n \rangle\}$ samples
$F \leftarrow \{f_1, ..., f_m\}$ features ▷ Samples may consist of more than the features here
$l_* \leftarrow$ default label ▷ If calling this for the first time, use $l_* \leftarrow$ MostCommonLabel(X)

if $X = \{\}$ **then**
    return a leaf node labelled with l_*
else if all samples in X have the same label l **then**
    return a leaf node labelled with l
else if $F = \{\}$ or all samples in X have the same value $\vec{x}$ but don't all have the same label **then**
    return a leaf node labelled with MostCommonLabel(X)
else ▷ Time to pick a feature and build a node based on it
    $f \leftarrow$ ChooseFeature(F, X)
    tree $\leftarrow$ a new nonleaf node whose query feature is f
    for each value v that f may take on **do**
        $X' \subseteq X$ those samples $\langle \vec{x}^{(i)}, l_i \rangle$ whose value $\vec{x}_f^{(i)} = v$
        child $\leftarrow$ BuildDecisionTree($X', F - \{f\}$, MostCommonLabel(X))
        Add *child* as a child subtree of *tree*, with a query of $\stackrel{?}{=} v$
    return *tree*

3.2.1 A Smart ChooseFeature Function

The compactness of the tree (and, in most cases, its generalization performance) is largely a function of how we implement the ChooseFeature function. This function weighs various features and selects one to build a nonleaf node. Why would one feature be preferred over another? Generally speaking, you want a feature which, after queried, won't require us to make many further queries before we can ascertain the label of a sample. How can we do this? Here's a rule of thumb for you: if you

[42] If you think about it, this makes sense: let's say we had a samples consisting of men and women (the labels). 99% of them are men. We chose to ask them: *what is your party affiliation?* It turns out that none of them are, say, members of the Green party. What do we do with a future sample which might be a Green party member and happened to wind up here? We assume he's a man.

[43] Often this is known as the **majority value**, which makes no sense at all, since the label doesn't have to have the majority of samples. For example, if the labels are *Bob, Bob, Mary, Sue* and *John*, the most common label is *Bob*.

don't want many further queries, you want a feature which has *already* broken your samples up into nice groups, each of which is relatively homogeneous with regard to their labels.

For example, let's say we have a set of samples with labels AAAAAAAABB. We're considering a feature f with three possible values, and so it'll make a nonleaf node in the tree with three children. Imagine these three values broke our set up into AAAA, AAAA, and BB. That'd be great! Totally homogeneous groups. Then the three children would just be leaf nodes labelled "A", "A", and "B", and we'd be done. Nice small tree.

But what if f broke the samples into AAAA, AAA and BBA? Not all hope is lost here: the AAAA and AAA groups would be just a single leaf node each. The BBA group would hopefully require just one more nonleaf node (with some other feature) to break things up. So although the groups aren't *totally* homogeneous, they're *nearly* homogeneous and that's probably pretty good.

Okay, what about AAAAAAA, BA, and BA? Now we have two groups which require extra nonleaf nodes. But here's the thing. While the majority of the groups are highly heterogenous, *most samples* will go into the AAAAAAAA group, which is homogenous. Sure the tree's a bit bigger, but we've identified a single label for the majority of the samples.

Put another way, imagine if you're trying to distinguishing between dogs and birds. One of your features is "number of feet". You have two million dogs for which the answer is '4", and a fair number for which the answer is "3". These are entirely homogeneous groups (all dog). And you have one million birds for which the answer is "2 or 1", with a smattering of dogs in there two sadly: but by and large homogeneous (bird). And then there's the "0" group, for which there are a *few* dogs and *few* birds, but which is very heterogeneous. Is this a good distinguishing feature? You bet it is! While there is a group which isn't at all homogeneous ("0"), it is very rare. The *huge majority* of dogs and birds have been distinguished from one another with a single question.

In general, what we're looking for is any f which results in a *very high proportion of samples* falling into *largely homogenous groups*.

First we need a way to measure the heterogeneity (sometimes called *impurity*) of a group. Let's say that we have n different kinds of labels and a set of samples X. If we divided X up by label, we'd have groups $X_1 ... X_n$. Based on these samples we estimate the *probability* of a sample falling into a certain label as being $P(1) = \frac{|X_1|}{|X|}, P(2) = \frac{|X_2|}{|X|}, ..., P(n) = \frac{|X_n|}{|X|}$. These probabilities must of course sum to one. We are looking for a heterogeneity estimation function H(P(1), ..., P(n)). Clearly if most of the probabilities are near 0.0, and one of them is close to 1.0, that's very homogeneous (so we want H to return a low score). On the other hand, if all the probabilities are uniformly even, that's pretty heterogeneous) and we want H to return a high score).

Two popular functions which exhibit this property are the **Information Function**[44]....

$$I(P(1),...,P(n)) = -\sum_{i=1}^{n} P(i)\lg P(i) \qquad \text{(If } P(i) = 0\text{, then assume } P(i)\lg P(i) = 0\text{)}$$

... and the **Gini Coefficient**...

$$G(P(1),...,P(n)) = 1 - \sum_{i=1}^{n} P(i)^2$$

You might take some time to convince yourself of this.

[44]Information theory is a mathematical field invented by Claude Shannon in the '50s. Shannon worked for AT&T and was interested in knowing how you could jam the most "information" into the smallest space, for compression purposes. Consider the two strings:

```
000000000000000000000000011111111111111111111111100
011010100010100010100010000010100000100010100010000
```

The first string might be said to have "less information" than the other string. We could create a compression technique, like run-length encoding, which can jam the first string into just a few bytes. But the second string appears to have too much randomness: thus we'd have to spend more bytes describing it. Often we think of the amount of information in a string as being correlated with how *statistically random* its data appears to be. Very often things that appear highly ordered have *low* information value, whereas items that are not very well ordered have *high* information value.

Shannon was a little more specific than this. However, he said that the information in a set of objects (the bits above, for example) is with respect to **some question Q** that we are asking. The information of a set **E** with regard to the question **Q** is roughly the number of *additional* questions we'd have to ask *after* asking question Q before we could nail down exactly the features of a given object in the set.

In the example above, imagine if the question **Q** is "are you in the first 25 bits?" and the remaining questions also asked location information. It wouldn't take long to nail down whether or not a given bit was a 1 or a 0 in the first situation — indeed, if the answer was "yes", you'd immediately know the bit was a 0. But in the second string, you'd have to ask a lot more position-oriented questions.

But what if the question **Q** was "are you in a prime or a non-prime position?" Here it turns out that the second string has 1's in exactly the prime positions, and 0's in exactly the non-prime positions — you'd need *no* additional questions! But you'd need a lot of additional questions to nail down the bit value in the first string. So don't be fooled by the *apparent* randomness of a set. What matters in information theory is *what questions you are able to ask*.

Shannon asked: if we ask question **Q**, and this breaks our set S into n subsets $S_1...S_n$ based on the possible answers $A_1...A_n$ to the question **Q**, how many more questions on average do we have to ask before we find the element we're looking for? Shannon figured that for each subset S_i we'd have to do a binary search, which for any subset S_i is going to take $O(\lg|S_i|)$ time. Then once you asked the **Q** question, the average number of additional questions you'd have to ask would be the sum of the various times of each S_i, weighted by the probability that A_i would be the answer to **Q** in the first place. This is:

$$\sum_{i=1}^{n} \mathrm{P}(A_i)\lg|S_i|$$

We use our samples as our estimate of $P(A_i)$, so $\mathrm{P}(A_i) = \frac{|S_i|}{|S|}$. So we can rewrite the equation as:

$$\sum_{i=1}^{n} \mathrm{P}(A_i)\lg|S_i| \quad = \quad \sum_{i=1}^{n} \mathrm{P}(A_i)\lg(|S|\mathrm{P}(A_i)) \quad = \quad \sum_{i=1}^{n} \mathrm{P}(A_i)\lg|S| + \mathrm{P}(A_i)\lg\mathrm{P}(A_i) \quad = \quad \lg|S| + \sum_{i=1}^{n} \mathrm{P}(A_i)\lg\mathrm{P}(A_i)$$

Now, if we *didn't* ask the question **Q** first, we'd have to do a binary search through just the set S, which is $O(\lg|S|)$. So how much time did asking question **Q** buy us? That's:

$$\lg|S| - (\lg|S| + \sum_{i=1}^{n} \mathrm{P}(A_i)\lg\mathrm{P}(A_i)) \quad = \quad -\sum_{i=1}^{n} \mathrm{P}(A_i)\lg\mathrm{P}(A_i)$$

One little note: What if $\mathrm{P}(A_i) = 0$? Then we have $0\lg 0$, which is $(0)(\infty)$. We'll define this to be equal to 0.

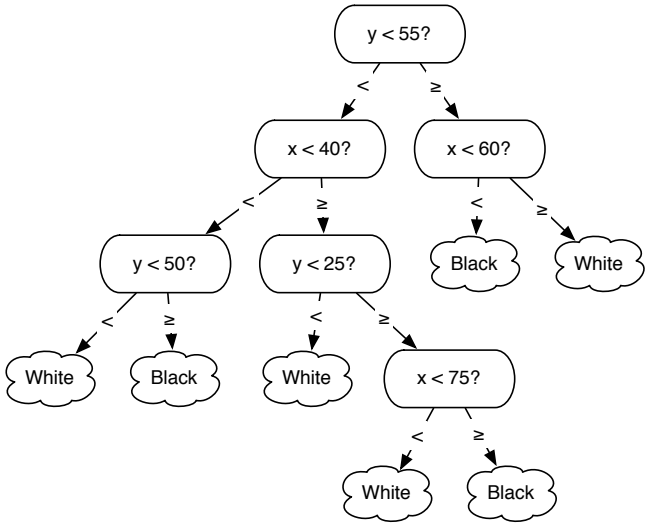


Figure 16 The metric decision tree responsible for divvying up the space as shown in Figure 17. Though in this example it so happens that the variable queries alternate between x and y, that need not be the case at all. Decision trees can also of course be a mixture of metric and categorical queries.

How do we use an impurity function to help us in decision trees? What we want is for a feature f to break our sample into groups such that the *average impurity* is as small as possible. We call this the **remainder** left over after we've broken our sample X up, using f, into groups $X_1, ..., X_m$.

$$\text{Remainder}(f, X) = \sum_{j=1}^{m} \frac{|X_j|}{|X|} \text{Impurity}(X_j)$$

Now we can define a *smart* **ChooseFeature** procedure as follows:

Algorithm 6 *ChooseFeature*

1: $X \leftarrow \{\langle \vec{x}^{(1)}, l_1 \rangle, ..., \langle \vec{x}^{(n)}, l_n \rangle\}$ samples
2: $F \leftarrow \{f_1, ..., f_m\}$ features

3: **return** the feature $f \in F$ where Remainder(f, X) is lowest, breaking ties arbitrarily

Why bother with all this? Because picking the lowest-information feature tends to break S into subsets that are the *closest to already being finished*. And *that* tends to produce smaller trees (because we don't have to ask as many questions later on).[45]

[45] As it turns out, this is only a heuristic: it doesn't always work. There are trees which are sometimes smaller than the information-theory technique would predict. And in fact over the space of *all* possible decision trees no **ChooseFeature** heuristic works better than (believe it or not) picking features totally at random! But over the space of decision trees that we usually use in the real world this heuristic tends to do a very good job.

3.3 Metric Features

Not all features are categorical. What if we have a metric feature like *temperature* or *age*?[46] It turns out we can do the same basic trick. But instead of creating n subtrees, one for each possible answer to the query, we'll create queries of the form: "is the temperature higher than 57 degrees?" or "is the age higher than 30?". This gives us *two* possible answers, and so we'll just have two subtrees.

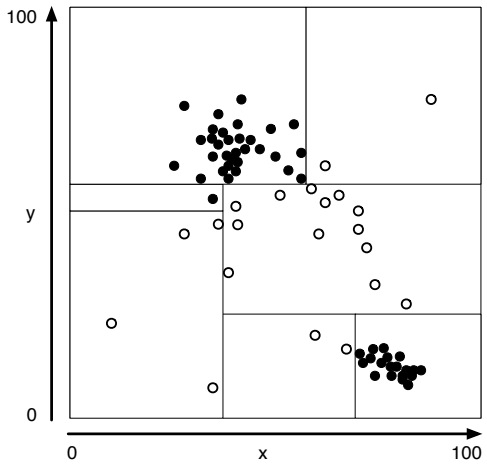


Figure 17 Samples in a metric space with two labels (black and white) divvied up into regions by the decision tree shown in Figure 16.

If you think about it, at any point in time a node is asking a query with regard to a certain remaining region of the space. And asking queries like this essentially slices that region in two, divided by a plane parallel to the dimension of the feature we're asking about. Thus in an important sense, decision trees may be thought of as cutting the space up into boxes or rectangles. Consider the decision tree in Figure 16, which has broken the two-dimensional metric space shown in Figure 17 into rectangles. Remember that there's **absolutely no reason** why you can't have a mixture of categorical and metric features (or other kinds of features for that matter, like toroidal ones). But in this example both features happen to be metric. There's also no reason why you can't have more labels than the two in this example: black and white).

This should give you a clue as to what kinds of classification problems are good for decision trees and which are bad. What if your classification regions aren't rectangular, but are curvy (for example)? You won't be able to *ever* nail down exactly the regions if the only dividers you have in your toolbox are right-angle lines and planes (see Figure 18).[47] This is part of the **learning bias** of the decision tree model. Every machine learning algorithm produces a model with some kind of bias (assumptions) in its representation which makes it good for some problems but poor for others. You'd like to choose a machine learning algorithm with a model whose bias matches your situation.

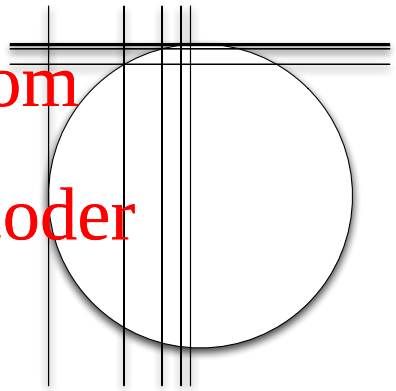


Figure 18 Squaring the circle. If your feature space is continuous, and your classification region is circular, you'll be asking an awful lot of a decision tree to define exactly which points are inside the region and which are outside.

This bias towards rectangular regions, be they in classification or metric space, does give the decision tree one interesting and useful capacity: since you're not drawing lines based on *multiple* features at one time, you can consider each of them independently of the other. Notably, you wont have to **scale** features so certain features don't dominate other features with their crazy values. And you can easily use mixed collections of boolean, categorical, and metric features in your feature space without having to assign categories unique numbers or consider how much angle is worth so much distance.

In the question "is the age higher than 30?", the value 30 is the **pivot**. Assuming we've picked a feature, what pivot should we choose? After all, there are an infinite number of possible pivot

[46] We'll assume all such features are single-dimensional. If you have a multidimensional metric feature like *coordinate location*, break it into its constituent parts (*x coordinate location* and *y coordinate location*, say).

[47] There exist decision trees which permit diagonal lines: they'll still not help you with a circle.

values, right? Well, not exactly. There are only a finite number of *samples*. So picking exotic pivots doesn't help us much: we'll pick pivots in-between the values those samples take on.[48]

For example, imagine if we had seven samples, each with the following values for f:

-2 4 7 7 9 15.3 25

This yields pivot points between -2 and 4 (half-way would be 1.0); between 4 and 7 (3.5); 7 and 9 (8); 9 and 15.3 (12.15); and 15.3 and 25 (20.25). Five pivot points in all. It's not helpful to pick pivot points before -2 or after 25, nor is it helpful to pick between 7 and 7.

Now the procedure is simple. For each pivot point, we divide the sample into two: those whose values are $<$ the pivot point, and those whose values are $\geq$ the pivot point.[49] Let these subgroups be called[50] $L \subset X$ and $G \subset X$. Since we only have two groups for our feature and pivot combination, we can define a Remainder (we'll call it "MetricRemainder") as follows:

$$\text{MetricRemainder}(f, p, X) = \frac{|L|}{|X|}\,\text{Impurity}(L) + \frac{|G|}{|X|}\,\text{Impurity}(G)$$

... where X is broken up into G and L based on the pivot query. That is: the Remainder is computed as usual, but with two splits rather than n splits. Our goal now is to pick the pivot with the lowest such remainder. So we make a simple pivot-choosing function along these lines:

Algorithm 7 *ChoosePivot*

1: $X \leftarrow \{\langle \vec{x}^{(1)}, l_1 \rangle, ..., \langle \vec{x}^{(n)}, l_n \rangle\}$ samples
2: $f \leftarrow$ feature

3: $Y \leftarrow \{\vec{x}^{(1)}, ..., \vec{x}^{(m)}\}$ sorted by f, smallest to largest, with duplicates (in f value) removed
4: $P \leftarrow \{p_1, ..., p_{m-1}\}$ pivots, where $p_i = \left(\vec{x}_f^{(i)} + \vec{x}_f^{(i+1)}\right)/2$ ▷ The i in $\vec{x}^{(i)}$ reflects its sorted position
5: **if** P is empty **then** ▷ Uh oh

6: **return** □ ▷ We'll test against this case later.
7: **else**
8: **return** the pivot $p_i \in P$ where MetricRemainder(f, p_i, X) is lowest, breaking ties arbitrarily

We have to verify prior to calling this function that there *are* pivots! This entails only calling the function if there is more than one sample, and that not all the samples have the same value for f.

Anyway, we can now revise ChooseFeature(...) to include both metric and categorical features:

Algorithm 8 *ChooseFeatureExtended*

1: $X \leftarrow \{\langle \vec{x}^{(1)}, l_1 \rangle, ..., \langle \vec{x}^{(n)}, l_n \rangle\}$ samples
2: $F \leftarrow \{f_1, ..., f_m\}$ features

3: **return** the feature $f \in F$ where Remainder(f, X) (if categorical) or
MetricRemainder$(f, \text{ChoosePivot}(X, f), X)$ (if metric) is lowest, breaking ties arbitrarily

[48] Usually exactly half-way in-between. But not always: in some of my research it's been necessary to pick other values.
[49] Or $\leq$ and $>$ if you like. Just be consistent.
[50] L for "less than", G for "greater than or equal". I dunno, sounded reasonable.

Now we can finally include both metric and categorical features in a revised decision tree builder. The primary difference in this function is that if the feature chosen is metric, we determine the best pivot for it, then break the samples into two sets according to the pivot, and recurse on those samples. Note: since we don't remove metric features when we recurse (unlike the categorical case), we need to check at the top of the function for metric features without valid pivots.[51] (Meditate deeply upon this.)

So here's the function.

Algorithm 9 *BuildDecisionTreeExtended*

$X \leftarrow \{\langle \vec{x}^{(1)}, l_1 \rangle, ..., \langle \vec{x}^{(n)}, l_n \rangle\}$ samples
$F \leftarrow \{f_1, ..., f_m\}$ features ▷ Samples may consist of more than the features here
$l_* \leftarrow$ default label ▷ If calling this for the first time, use $l_* \leftarrow$ MostCommonLabel(X)

$G \leftarrow$ all metric features in $f \in F$ for which ChoosePivot$(f, X) = \square$
$F \leftarrow F - G$ ▷ We get rid of metric features that have no valid pivots
if $X = \{\}$ **then**
    return a leaf node labelled with l_*
else if all samples in X have the same label l **then**
    return a leaf node labelled with l
else if $F = \{\}$ or all samples in X have the same value $\vec{x}$ but don't all have the same label **then**
    return a leaf node labelled with MostCommonLabel(X)
else ▷ Time to pick a feature and build a node based on it
    $f \leftarrow$ ChooseFeatureExtended(F, X)
    tree $\leftarrow$ a new non-leaf node whose query feature is f
    if f is a categorical feature **then**
        for each value v that f may take on **do**
            $X' \subseteq X \leftarrow$ those samples $\langle \vec{x}^{(i)}, l_i \rangle$ whose value $\vec{x}_f^{(i)} = v$
            child $\leftarrow$ BuildDecisionTreeExtended$(X', F - \{f\}$, MostCommonLabel$(X))$
            Add *child* as a child subtree of *tree*, with a query of $\stackrel{?}{=} v$
        return *tree*
    else ▷ Continuous feature
        $v \leftarrow$ ChoosePivot(f, X)
        $X' \subseteq X \leftarrow$ those samples $\langle \vec{x}^{(i)}, l_i \rangle$ whose value $\vec{x}_f^{(i)} < v$
        left $\leftarrow$ BuildDecisionTreeExtended$(X', F,$ MostCommonLabel$(X))$
        Add *left* as a child of *tree*, with a query of $\stackrel{?}{<} v$
        $X' \subseteq X \leftarrow$ those samples $\langle \vec{x}^{(i)}, l_i \rangle$ whose value $\vec{x}_f^{(i)} \geq v$
        right $\leftarrow$ BuildDecisionTreeExtended$(X', F,$ MostCommonLabel$(X))$
        Add *right* as a child of *tree*, with a query of $\stackrel{?}{\geq} v$

[51] One way you *could* "remove" a metric features is by considering a "feature" to be a feature plus a pivot point. Thus if you've used a given pivot point for a given feature, you won't use it again; and if all the pivot points are used up, the feature is considered to be used up.