CS181 Lecture 6 — Decision Trees

Today we turn to a new topic: supervised learning. The problem is to learn a concept from a bunch of training examples, some of which satisfy the concept, and others that don't. This is a classical machine learning problem. After discussing some basic ideas for supervised learning, we will turn to a particular learning algorithm, decision trees.

Optional readings for the next four lectures: Chapter 18 of Russell & Norvig, Chapters 1 & 3 of Mitchell "Machine Learning"

1 The Task

Whenever we talk about learning, there is a hierarchy of tasks to consider. We must first of all talk about the ultimate task to be performed, the thing we are trying to learn to do. Then we can talk about the task of learning how to perform the ultimate task. Finally, we can consider the task of designing a learning algorithm.

For the next few classes, we will focus on classification as the ultimate task to be performed. Classification means determining what category an object falls into, based on its features. For example, we might try to classify a plant

as nutritious or poisonous, based on biological features such as color, leaf shape and so on. Or we might try to classify a pixel image as being a particular digit. The tasks fall into a hierarchy as follows:

Classification An object is described by a set of attributes or features $X = X_1,...,X_n$. There is a set C of possible classes $C_1,...,C_m$. Given the features x of a particular object, a classifier needs to determine its class c.[1] Thus a classifier is a function $f : X \rightarrow C$.

For a particular datum with features x and class c, the performance of the classifier is measured by the standard utility function, which is 1 if f(x) = c (correct classification), 0 otherwise. However, as we discussed, a richer utility function may be appropriate in some domains. E.g., in a medical domain, the cost of false negatives is higher than that of false positives. We will stick to the simple utility function in this class.

Supervised Learning The goal of supervised learning is to learn a classifier from a set of labeled training examples. Each training example D_i consists of attribute values $x_{i\,1},...,x_{i\,n}$ and a class $c_{i.2}$ A supervised learning agent takes a training set $D = D_1,...,D_N$ and outputs a function $f : X \rightarrow C$. Thus a supervised learning algorithm is a function L from labeled training sets to classifiers. For a particular training set D, L(D) is a classifier.

How do we evaluate the performance of a learning algorithm L on a training set D? Since the goal is to produce a classifier, we measure the performance of the learning algorithm through the performance of the classifier. That is, the utility earned by the learning agent L is the expected utility the classifier L(D) will achieve on future examples. A key point is that the classifier's performance is measured relative to future examples that it has not yet seen, and not the training set. The critical question is whether the learning algorithm is able to produce classifiers that generalize to data that it did not see.

[1] We will generally use capital letters such as X and C to denote variables, and small letters to denote particular values for the variables. Also, we use boldface to denote vectors or sets.

In order to define the performance measure, we need a probability distribution over unseen instances. I.e., we have a probability distribution over < X,C >, the features and class of an unseen instance. The expected performance of a classifier f on unseen instances is then P(f(x) = c), the probability that it will predict the correct class c given the features x that it sees.

The probability distribution over unseen instances represents "ground truth", the way the world actually is. Normally, we do not have access to ground truth. We can only estimate this distribution using a test set of labeled instances. When we use a test set, the performance of the classifier produced by a learning algorithm is measured by the fraction of the test set that it classifies correctly.

Designing a Learning Algorithm When we have to choose a particular learning algorithm for a particular domain, we do not generally know exactly what the correct function will be, or exactly what the training set will look like. We need to design the learning algorithm so that it will perform well in the domain, given the particular characteristics of the domain, and given the amount of training data it will get. For example, we need to choose the features for the algorithm. Algorithms may have parameters that we need to set. Different algorithms are good at learning different kinds of functions. Also, as we will see, some algorithms need a lot of data to work well. A key aspect of studying machine learning is not only to understand the learning algorithms themselves, but to understand what aspects of the domain make them work well.

Often, the possible classes in a classification problem will be true and false. In this case, the learning task is sometimes called concept learning, because the task is to learn a function that determines whether or not a given object is an instance of a concept based on its features. I.e., the learned function is the definition of a concept. For concept learning, the training data consists

[2]We will generally use superscripts to index data items.

of positive and negative examples: instances that are or are not examples of the concept. If the attributes themselves are also Boolean, then the problem is that of learning Boolean concepts. In this case, the concept is defined by a Boolean formula over the attributes. We will focus mostly on the case of learning Boolean concepts, because they are simple, but already cover most of the issues that come up.

There are some useful distinctions we can make concerning learning problems. The first concerns whether the classification task is deterministic or non-deterministic. This is a distinction concerning ground truth. In a deterministic domain, if two data instances have the same features, then they necessarily have the same classification: in symbols $x_i = x_j \Rightarrow c_i = c_j$. In an non-deterministic domain, two instances may have the same features but different classifications. A non-deterministic domain is often called noisy. Noise can happen either because of inherent non-determinism in the domain, or because of errors and mislabeling in the data.

The second distinction is similar, but concerns the training data itself. The training data is consistent if there are no two instances that have the same features but different classifications, otherwise it is inconsistent. Clearly, if the domain is deterministic then a training set for that domain will be consistent, but the converse does not hold.

2 Is Learning Possible?

Before we go on to consider how learning is done, we need to consider whether learning is possible at all. In fact, there are strong arguments that learning is logically impossible!

Consider the problem of learning Boolean concepts, and assume the target concept is deterministic. We want to learn a Boolean formula over the attributes $X_1,...,X_n$ from a training set consisting of positive and negative examples. For notational convenience, let's order the training set so that the positive examples appear first. So the data consists of positive examples $D_1,...,D_k$ and negative examples $D_{k+1},...,D_N$.

Suppose we want to classify a new instance based on its features X. What should we do? There are two cases. If X appears in the training set, we should provide exactly the same classification. If it does not, then one can argue that the training set provides absolutely no information for this case! There are at least two concepts consistent with the training data. The first says that only the positive examples seen in the training set are actual examples of the concept. Symbolically, this is the concept

$$(x_{1\,1} \wedge ... \wedge x_{1\quad n}) \vee ... \vee (x_{k\quad 1} \wedge ... \wedge x_{k\quad n})$$

The second concept says that only the negative examples seen in the training set are not examples of the concept. Symbolically,

$$\neg\{(x_{k+1\,1\quad} \wedge ... \wedge x_{k+1\ n\quad}) \vee ... \vee (x_{N\quad 1} \wedge ... \wedge x_{N\quad n})\}$$

The first concept is the most specific concept consistent with the training set: it rules out all instances not implied by the training set. The second concept is the most general concept consistent with the training set: it includes all instances not ruled out by the training set. Our new unseen instance will fall under the second concept and not the first.

The conclusion of this argument is that, logically speaking, the training set tells us nothing about unseen instances. We can only classify instances that we have already seen in the training set. Thus, we are only able to memorize the training set, not to generalize to unseen instances. And, the argument concludes, since learning requires generalization, learning is logically impossible!

This argument is a form of the famous induction problem that has been extensively studied in philosophy. There are two other forms of the induction problem that philosophers worry about.

The classical induction problem, due to Hume, is that learning relies on the fundamental assumption that the future resembles the past. But how do we know this? This form of the induction problem is even sharper than the one we described above, because it says that even memorizing the data is useless. Despite its enormous philosophical importance, we will ignore this form of the induction problem, and simply assume that the past is a reliable guide to the future. Philosophers don't actually doubt this; rather they ask why it is true, and how can we know that it is true. These are not practical concerns when it comes to machine learning.

A more recent induction problem, due to Goodman, is as follows. Even if we grant that the future resembles the past, the past can imply many things about the future. Let us define the term "grue" to mean anything that is green before January 1st 2100 and blue thereafter. Suppose we have observed that the grass is green today. Can we expect that it will be green in a hundred year's time? We would say yes — grass has always been green, and we can reason inductively that it will continue to be green. But, so the argument goes, grass has always been grue, so by the same reasoning we should conclude that it will continue to be grue, in which case it will not be green in a hundred year's time. So induction implies equally that grass will and will not be green in 2100.

We will not try to resolve Goodman's gruesome puzzle here. Rather, we will draw the moral that the way in which we frame a learning problem and the choice of features is crucial to whether or not learning is actually possible. We will assume in all of our work that a set of features has been chosen that actually characterize a particular instance, and not some external attribute such as time.

Our induction problem is somewhat different from these. It says that even

if we assume that the past is a reliable guide as to the future, and that we have framed the past in the right terms so that it is a reliable guide, we can still, logically, do no more than memorize the past. So how is learning possible?

3 Inductive Bias

The answer is that in fact, for learning to be possible, we need to make some assumptions. These assumptions are called our inductive bias, or simply bias. They will cause us to fall into error in some cases, but we must allow for that in order to learn. The argument from the previous section is sometimes called the No Free Lunch theorem, because learning without the possibility of error would be a free lunch, and there is no such thing.

There are two basic kinds of inductive bias. The first is called restriction bias because the learning algorithm does not consider all possible classifiers, only those that fall in a certain class. The set of classifiers that the algorithm considers is called the hypothesis space of the algorithm, and the different possible classifiers are called hypotheses. An example, is an algorithm for learning Boolean concepts that only considers conjunctive concepts. A conjunctive concept is formed from the conjunction of literals, where a literal is a single attribute or its negation. Thus $X_1 \wedge \neg X_2$ is a conjunctive concept, but $X_1 \vee \neg X_2$ is not.

An algorithm that uses a restriction bias is able to learn. There may be only one hypothesis in its hypothesis space H consistent with the training data, in which case it returns that hypothesis. It is also possible that no hypotheses in H are consistent with the training data, even if the training data is consistent. In that case, the learning algorithm may be able to return the hypotheses that best matches the data, even though the match is imperfect.

The second kind of bias is called a preference bias. A preference bias indicates a relative ranking between different hypotheses the algorithm considers. For example, in learning Boolean concepts, one may prefer more specific or

more general hypotheses, in which case the first or second formula above will be returned. An algorithm with no restriction bias that does have a preference bias is able to learn. For a consistent data set it will choose the most preferred hypothesis consistent with the data. For an inconsistent data set, it will trade off the preferability of a hypothesis with the degree to which it agrees with the data.

There are two basic ways of achieving a preference bias. One is to define a scoring function that assigns a score to different hypotheses, so that H_1 is preferred to H_2 if it has a higher score. The second is to define a search process that searches through the hypothesis space in a particular way, looking for hypotheses that fit the data well. Hypotheses that appear earlier in the search process are automatically preferred to those that appear later.

Understanding its inductive bias is the most important thing to know about any learning algorithm. When considering an algorithm for a particular problem, it is critical to ask whether or not its inductive bias actually fits what you believe to be true about the domain. Is the true concept actually in its hypothesis space, (or at least, is there a hypothesis in its hypothesis space that is close to the truth)? Does its preference bias lead it to prefer hypotheses that are appropriate for the domain?

Different algorithms have stronger or weaker bias. Algorithms with weaker bias require more training data to learn well. Does the algorithm have the right

amount of bias for the training data that is available?

4 Decision Trees

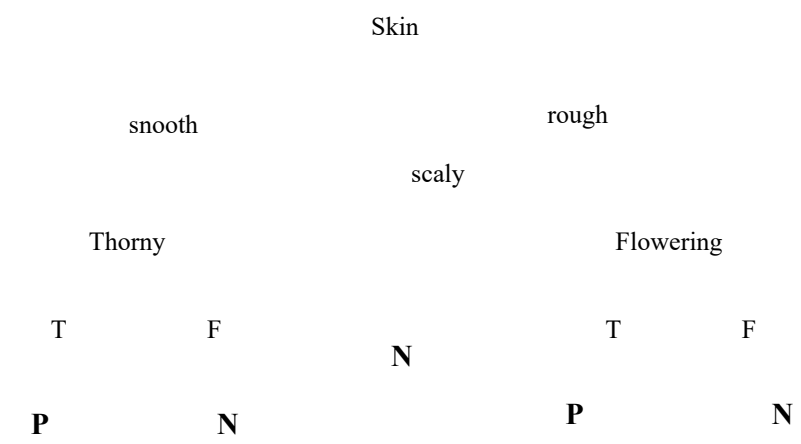
Given these concepts, we are ready to plunge into studying decision trees. The decision tree representation and learning algorithm are quite simple, yet they are widely used in practice. The combination of simplicity and general applicability make them a natural choice to consider for many domains. In describing the decision tree framework, we need to describe two things: the decision trees themselves, which are the classifiers to be learned, and the algorithm for learning a decision tree from data.

A decision tree is a representation of a function from a set of discrete attributes to a classification. As its name suggests, a decision tree is a tree. Each internal node of the tree is labeled by an attribute X_i. There is a branch leaving the X_i node corresponding to each possible value of X_i. Leaves of the tree are labeled by possible classifications.

For example, let us consider the Nutritious vs Poisonous classification problem.[3] Suppose there are four attributes with the following values:

- Skin (smooth, rough or scaly)
- Color (pink, purple or orange)
- Thorny (true or false)
- Flowering (true or false)

A possible decision tree for this domain is as follows:



[3]This example is based on Chapter 3 of Mitchell

A decision tree classifies an instance in a very simple way. Beginning at the root, it goes through the tree, selecting the appropriate subtree based on attribute values of the instance, until it reaches a classification at a leaf. Formally, the tree T classifies the instance x as follows: if the root of T is a leaf c, the classification is c; otherwise, the root is an attribute X_i, and the classification is the classification of the subtree reached by the branch labeled x_i.

What is the expressive power of decision trees? In fact, any function from discrete attributes X to classification C can be represented as a decision tree. To see this, consider the complete tree, which splits on each of the attributes in turn, so that a leaf is reached only after assigning values to all the attributes. Any function from X to C can be represented with such a tree.

This means that a learning algorithm that learns decision trees does not have a restriction bias. Therefore, if the algorithm is to be successful, it must have a preference bias. What kind of preference bias is appropriate here? The preference bias of decision tree learning algorithms is to prefer shorter trees. This bias is a form of Occam's razor: if there are multiple ways of explaining the same phenomenon, the simpler hypothesis is to be preferred. In the case of decision trees, simpler means shorter.

Because of this preference bias, decision trees are naturally better at expressing some kinds of functions than others. They are very good at representing functions where a small number of attributes are critical to the classification. They are bad at functions that depend approximately equally on all attributes. Examples of functions which decision trees have a hard time with are the majority and parity functions. Again I want to stress the key point that understanding the inductive bias of an algorithm can tell us what sort of domains it will work well in.

5 Learning Decision Trees

We have now described the hypothesis space of decision trees, and the preference bias. How do we actually go about learning decision trees? One obvious answer is to consider all possible trees, and choose the shortest one that is consistent with the training data. In the case of inconsistent training data, there will be no tree that is consistent with it, Nevertheless, one may choose the shortest tree amongst those with fewest errors on the training data.

This is clearly an infeasible approach. The number of decision trees of depth d is doubly exponential in d (why?). So an exhaustive search, even to a shallow depth, is clearly not going to work.

Instead, a greedy search procedure is used, that grows a decision tree one node at a time, from the root downwards. The learning algorithm, called ID3, is recursive. It takes as arguments a set of training instances D, and a set of possible attributes X to split on. It makes a decision what to do at the root of the tree. There are basically two possibilities:

- Stop growing the tree, and return a classification.

- Split on an attribute X in X. In this case, the training data is split into subsets D_x, corresponding to the different possible values x of X. A subtree is recursively grown for each of D_x, and the edge from the root to the subtree D_x is labeled by the corresponding value x. For the recursive call, X is removed from the set of available attributes to split on, since there is no point in splitting on the same attribute twice.

The algorithm is as follows:

```
ID3(D, X)
    Let T be a new tree
    If all examples in D have the same class c
        Label(T) = c
        Return T
    If X = ∅
        Label(T) = the most common classification in D
        Return T

    Choose the best splitting attribute X in X // described below
    Label(T) = X
    For each value x of X
        Let Dx be the subset of instances in D that have value x for X
        If Dx is empty
            Let Tx be a new tree
            Label(T) = the most common classification in D
        Else
            Tx = ID3(Dx, X − {X})
        Add a branch from T to Tx, labeled by x
    Return T
```

Notice that ID3 stops growing the tree for one of three reasons:

- There is no data left. In this case, there is no need to grow the tree further, because any further elaboration would introduce complexity that is not supported by the training data. This is where the inductive bias of the algorithm comes into play.

- All training instances have the same classification c. In this case the

constant function that returns c can be returned as the perfect classifier, and there is no need to elaborate further

- There are no attributes left to split on. Since all attributes have already been split on, all instances in the training data must have the same value for all attributes, so there is no point in splitting any further. If this case happens, even though there is still some training data, and there is more than one classification in the data, then the data must necessarily be inconsistent (why?).

6 Information Gain

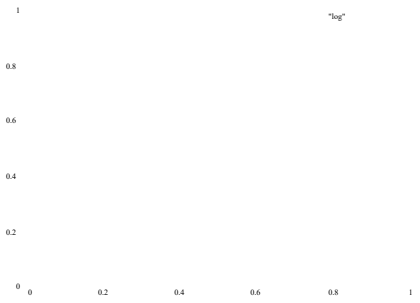
The one thing that remains to be defined is the criterion for choosing which attribute to split on. What criterion should we choose? Consider the ultimate goal: to produce short trees. ID3 is a greedy algorithm, that greedily tries to grow the tree in such a way that the result is a short tree. Therefore, the criterion for choosing which attribute to split on should tend to produce short trees. A general heuristic is that the more "orderly" the data is, the shorter the tree needed to represent it. At one extreme, if all instances have the same classification, we can use a tree of depth 0.

Information theory gives us a precise concept of "orderliness" of data. The key notion in information theory is entropy, defined as follows:

$$\text{Entropy}(D) = -\sum_{c \in C} \frac{N_c}{N} \log_2 \frac{N_c}{N}$$

where N is the total number of instances in D, and N_c is the number of instances that have classification c. Entropy can be interpreted as the amount of disorder in the data. If the data has high entropy, it is less orderly. One information theoretic interpretation is that the entropy is the number of bits per instance required to encode the data. For example, let us consider binary classification. At one extreme, if all instances are positive (or all negative), Entropy(D) = $-1 \log 1 = 0$. I can tell you that all the data is positive (or negative), and then I

don’t need to send you any information about individual instances. At the other extreme, half the instances are positive and half negative, and Entropy(D) = −1/2 log 1/2 − 1/2 log 1/2 = 1. In this case, I can do no better than telling you explicitly whether each individual instance is positive or negative. The graph of the entropy, as a function of the fraction of instances that are positive, looks like this:



Since entropy is a measure of the amount of information required to encode the data, and we want to encode the data in as compact a way as possible, we will choose an attribute to split on that provides us with the most information. We first define a function that tells us the amount of information that is still

required to encode the data, after splitting on an attribute X:

$$\text{Remainder}(X, D) = \sum_x \frac{N_x}{N} \text{Entropy}(D_x)$$

This encodes the expected number of bits needed to encode the data if we first test on attribute X. Now, we can define the information gain of an attribute as follows:

$$\text{Gain}(X, D) = \text{Entropy}(D) - \text{Remainder}(X, D)$$

Our criterion for choosing which attribute to split on is to choose the one with highest information gain. It is a basic information theoretic result that the information gain cannot be negative for any attribute. However, it can be zero, and an information gain of zero means that splitting on an attribute is useless and provides no additional information. If the information gain for all attributes is zero, it means that we should stop splitting. We therefore need to modify the ID3 algorithm slightly to provide one more termination condition. We replace the line

```
Choose the best splitting attribute X in X
```

with the lines

```
Let X be the attribute that maximizes Gain(X, D)
If Gain(X, D)=0
    Label(T) = the most common classification in D
    Return T
```