

CS221 Lecture notes #6

Decision trees

Last time, we discussed two supervised learning algorithms: linear regression and logistic regression. These algorithms worked well when our inputs x were continuous. Now we will discuss another classification algorithm, called decision trees, which is more appropriate for discrete inputs. For instance, suppose we are trying to predict if our roommate will eat a certain food. Our features are as follows:

1. HUNGER: starving, hungry, can-eat, or full
2. LIKE: yes/no
3. HEALTHY: yes/no
4. PRICE: free, cheap, expensive

Our target variable is $y \in \{0, 1\}$, which evaluates to 1 if our roommate eats the food.

How would we solve this problem using logistic regression? We might define a feature vector which has a $\{0, 1\}$ entry for each possible value of each feature. (In other words, if there are four features, each taking on three possible values, our inputs will be of length 12.) If a feature i takes the value v_i , we assign a 1 to the corresponding element of the input x ; otherwise, we assign it a 0. For an example where

$$x^{(i)} = (\text{hungry}, \text{yes}, \text{yes}, \text{cheap}),$$

our 11-dimensional feature vector might be

$$x = [0 \ 1 \ 0 \ 0 \ 1 \ 0 \ 1 \ 0 \ 0 \ 1 \ 0].$$

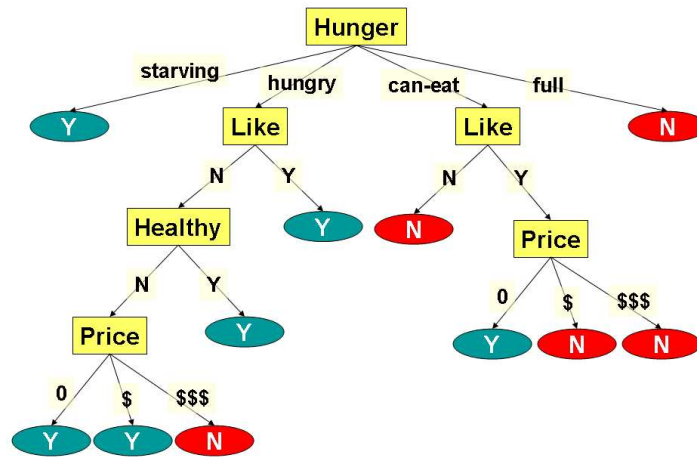


Figure 1: An example decision tree for deciding whether our roommate will eat a given food.

You can imagine this process producing enormous feature vectors if we use it in domains with large numbers of variables.¹

Instead, we can use a **decision tree** classifier. A decision tree for this domain might look like the one in Figure 1. Suppose we’re given the same example $x^{(i)}$ as above. We begin at the **root node**, which is labeled “Hunger.” Because our roommate is hungry, we descend down the branch labeled “hungry” to get to the node labeled “Like.” Since our roommate likes the food, we descend down the “Y” branch. Now we’ve arrived at a **leaf node**, which happens to give the answer “Yes.” Hence, we conclude that our roommate will eat the food. In general, the internal nodes of the tree will correspond to features, the edges will correspond to different values of the feature, and the leaves will correspond to yes/no predictions. Or, rather than a simple yes or no, we might associate with each leaf ℓ a probability p_ℓ , which is the probability that our roommate will eat the food in a situation associated with ℓ .

¹You may have observed that we can do slightly better by using only a single component of the inputs to represent binary features, but this won’t solve the basic problem of large feature vectors.

1 Decision tree learning

Now, we discuss how to learn a decision tree from data. Suppose we take careful notes on our roommate's eating habits and come up with the training data shown in Table 1. We define a scoring function ℓ as follows.² For a decision tree T , let $\ell_T(x)$ (or simply $\ell(x)$) be the leaf reached on input x . Then

$$P(y|x; T) = \begin{cases} p_{\ell(x)} & \text{if } y = 1 \\ 1 - p_{\ell(x)} & \text{if } y = 0 \end{cases} .$$

A more compact way of writing this is

$$P(y|x; T) = (p_{\ell(x)})^y (1 - p_{\ell(x)})^{1-y} .$$

So given a training set $\{(x^{(i)}, y^{(i)}), \dots, (x^{(m)}, y^{(m)})\}$, the data likelihood is:

$$\begin{aligned} \mathcal{L}(T) &= \prod_{i=1}^m P(y^{(i)}|x^{(i)}; T) \\ &= \prod_{i=1}^m (p_{\ell(x^{(i)})})^{y^{(i)}} (1 - p_{\ell(x^{(i)})})^{1-y^{(i)}} . \end{aligned}$$

As with our other maximum likelihood examples, our goal is to choose the tree's structure and parameters to maximize the log likelihood:

$$\begin{aligned} \ell(T) &= \log \mathcal{L}(T) \\ &= \sum_{i=1}^m y^{(i)} \log p_{\ell(x^{(i)})} + (1 - y^{(i)}) \log(1 - p_{\ell(x^{(i)})}) \end{aligned}$$

Now we have two tasks. We have to choose the structure of our tree, and then given the structure of our tree, we need to assign the probability p_ℓ to each leaf ℓ . It turns out that, given a tree structure, we can solve for the latter in closed form; so, our strategy will be to search over trees, using this closed-form solution to help evaluate our trees.

²We will overload notation in this lecture by using ℓ to represent both leaves of the tree and log likelihood.

Hunger	Like	Healthy	Price	Eat
full	y	n	free	n
starving	n	y	expensive	y
hungry	n	y	free	y
hungry	n	y	expensive	y
hungry	n	n	cheap	y
hungry	n	n	expensive	n

Table 1: Training examples for our food domain.

1.1 Assigning probabilities to leaves

Define S_ℓ to be the subset of the training examples which reach leaf ℓ . We partition our training set into disjoint subsets, one for each leaf. So

$$\begin{aligned} \ell(T) &= \sum_{i=1}^m \log P(y^{(i)}|x^{(i)}; T) \\ &= \sum_{\text{leaves } \ell} \sum_{(x^{(i)}, y^{(i)}) \in S_\ell} \log P(y^{(i)}|x^{(i)}; T). \end{aligned}$$

Now, to actually solve for p_ℓ for a particular leaf ℓ , note that the only terms in the above equation which are affected by p_ℓ are the ones associated with leaf ℓ . In other words, we can choose P_ℓ to maximize

$$\sum_{(x^{(i)}, y^{(i)}) \in S_\ell} \log P(y^{(i)}|x^{(i)}; T)$$

for the particular leaf ℓ . This is exactly like our example from the previous set of notes, where we computed the maximum likelihood estimate for ϕ for a Bernoulli(ϕ) random variable (e.g., coin flips). We saw in that case that the maximum likelihood estimate of ϕ was the proportion of flips which came up heads. In this case, we get

$$\begin{aligned} p_\ell &= \frac{m_{\ell+}}{m_{\ell+} + m_{\ell-}} \\ &= \frac{m_{\ell+}}{m_\ell} \end{aligned}$$

where $m_{\ell+}$ is the number of positive examples associated with leaf ℓ , $m_{\ell-}$ is the number of negative examples, and m_ℓ is the total number of examples which reach ℓ . Hence, our maximum likelihood estimate of p_ℓ is what we would expect it to be: the proportion of instances reaching that leaf which we labeled as positive.

```

DecisionTree(S)
   $X_i := \text{ChooseFeature}()$ 
  Make  $T$  the root of the tree, labeled with  $X_i$ .
  For each value  $v$  of the feature  $X_i$ ,
    Let  $T_v = \text{DecisionTree}(S_v)$ .
    Add  $T_v$  as a child of root node  $T$ .
  Return  $T$ .

```

Figure 2: Decision tree learning algorithm. S_v denotes the subset of the training set S where the feature X_i takes the value v . These are the only training examples which could possibly affect that child.

1.2 Learning the tree structure

We showed how to assign probabilities to leaves given a fixed tree structure, and now we need to actually find the structure. In principle, we could evaluate all tree structures and choose the one with largest likelihood, but we can't afford to because the number of trees is exponential. Instead, we are going to use a greedy hill-climbing algorithm to approximately find the best tree. Our hill-climbing approach will begin with a depth-zero tree (a single root node), and incrementally split a leaf node on each iteration. More formally, our local search problem will be as follows:

- State space: tree structures
- Initial state: tree composing only a root node (1 leaf)
- Operators: pick a leaf and split it.

We can describe our search algorithm with the procedure shown in Figure 2. This procedure chooses some feature X_i , and then for each possible value v which X_i could take, it adds a child for v by recursively calling itself on the subset of the training data where $X_i = v$. Now we just need to write `ChooseFeature`, i.e., choose which feature to split on next. We are going to choose this feature in a greedy manner: we will choose the split which gives the largest increase in the log likelihood ℓ .

We already know what the optimal values are for p_ℓ for a given tree structure. Specifically, we know that $m_{\ell+} = p_\ell m_\ell$. So we can express the log

likelihood for S_ℓ , the set of all training instances associated with a given leaf ℓ , in closed form:

$$\begin{aligned}\ell(S_\ell) &= m_{\ell+} \log p_\ell + m_{\ell-} \log(1 - p_\ell) \\ &= m_\ell [p_\ell \log p_\ell + (1 - p_\ell) \log(1 - p_\ell)] \\ &= -m_\ell \mathcal{H}(p_\ell),\end{aligned}$$

where \mathcal{H} is the **entropy function**, defined as

$$\mathcal{H}(p) = -p \log p - (1 - p) \log(1 - p).$$

Then the total log likelihood, the quantity we are trying to maximize, will be the sum of this quantity over all of the leaf nodes in our tree:

$$\ell(T) = \sum_{\text{leaves } \ell \text{ of } T} -m_\ell \mathcal{H}(p_\ell).$$

The entropy function is plotted in Figure 3. It has a maximum of 1 at $p = 0.5$ and decays to zero as p approaches 0 or 1. Intuitively, the higher the entropy of a random variable, the less we “know” its value. For instance, we “know” the value of a constant random variable (e.g., one which always equals 1), and so it has entropy zero. We “know” nothing about the outcome of a coin flip, so it has large entropy. From this intuition, it seems plausible that we would want to minimize entropy. To reiterate, choosing the split feature which maximizes likelihood is equivalent to choosing the one which gives the largest reduction in entropy. This criterion is often referred to as **information gain**.³

Suppose we are trying to choose a variable on which to split a leaf ℓ into two children, ℓ_1 and ℓ_2 , to get a new tree T' . How do we actually compute the information gain? The information gain is defined as the reduction in the total entropy achieved by the split, e.g.

$$\ell(T') - \ell(T) = \sum_{\text{leaves } \ell \text{ of } T'} -m_\ell \mathcal{H}(p_\ell) - \sum_{\text{leaves } \ell \text{ of } T} -m_\ell \mathcal{H}(p_\ell).$$

Fortunately, we don’t actually have to compute any sums over the entire training set. All of the leaves in T and T' are identical aside from the one

³You may ask, do we have to keep track of previously split variables? Actually, this is already handled by the information gain criterion. Splitting on a previously split feature can’t possibly reduce entropy, so such a feature will never be chosen.

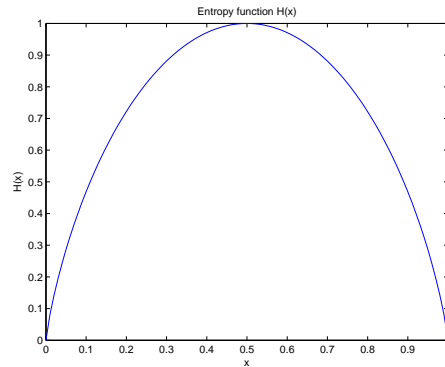


Figure 3: The entropy function $\mathcal{H}(x)$.

we just split. Therefore, the information gain can be computed entirely in terms of ℓ , ℓ_1 , and ℓ_2 :

$$m_\ell \mathcal{H}(p_\ell) - m_{\ell_1} \mathcal{H}(p_{\ell_1}) - m_{\ell_2} \mathcal{H}(p_{\ell_2}).$$

One final decision is when to stop splitting. We have several options:

- When all of the training examples agree at each leaf node. In this case, we can just return either 1.0 or 0.0 for each leaf node.
- When we have no more examples down a particular branch. We can then assign a default value, such as the overall fraction of positive examples in the entire training set, or the overall fraction of positive examples under the parent node.
- When we have no more attributes to split on. Then we let p_ℓ be the proportion of positive examples in that part of the tree.

2 Summary

Decision trees are one of the earliest machine learning algorithms. They are simple and fast and can be applied to a wide range of problems. Decision trees work fairly well, but not as well as some other state of the art algorithms. However, we will discuss two algorithms in the next lecture, bagging and boosting, which vastly improve the performance of decision trees.

In class, we also discussed the following examples of decision tree applications:

1. Silicon Graphics Company flight simulator. A decision tree “watches” (amateur) humans use the flight simulator, and learns to predict what move the human will make. Amazingly, the algorithm learned to fly *better* than the humans.
2. Elective C-sections. If something goes wrong during the birth process, the mother may be rushed off to the emergency room for an emergency C-section procedure which carries higher risk to the mother than an elective C-section. It is better for the woman to undergo an *elective* C-section, but hard to predict when one will be necessary. One group gathered a lot of medical data on patients and trained a decision tree to predict whether a woman will require a C-section. This algorithm even discovered a previously unknown class of high-risk women, for which their expected outcome can be significantly improved by undergoing elective C-sections.