Chapter 6

Decision Tree Learning

6.1 Classification problems in AI

Look at a photograph of an animal and determine if it is a dog, a cat, or a porcupine. Listen to a recording of a song and recognize if it is Beethoven, Bach or the Beatles. Analyze regional radar, temperature, barometric and wind reports and decide to carry an umbrella or not. These are all classification problems. Given vectors of input data, the output is an element of a small, discrete set, as in {dog, cat, porcupine} or {yes umbrella, no umbrella}. The input data can be discrete (as in the color of a pixel) or continuous (the air temperature).

A *classifier* is an algorithm that predicts the category of an observation. We will refer to observations as *feature vectors* and the categories as *classifications*. Feature vectors are comprised of *features* (also called *explanatory variables* or *independent variables* in other presentations). The classification is sometimes called a *response variable* or *dependent variable* or simply *output*. We assume there is a function f (sometimes called a *concept*) that maps feature vectors to correct classifications. A classifier outputs classifications based on its own approximation $\tilde{f}$ to f.

The correctness of $\tilde{f}$ can be measured relative to a *test set*, T, which is a subset of observations from the domain of all possible

observations. The *error rate* of $\tilde{f}$ on the test set is the fraction of $x \in T$ for which $f(x) \neq \tilde{f}(x)$.

A classifier *learning* algorithm is a type of *supervised learning* algorithm that has access to a *training set* of correctly classified observations and builds its own approximation $\tilde{f}$ of the true classification function f. The hope is that if $\tilde{f}$ exhibits a low error rate on the training set, then it will also result in a low error rate on any unseen test set.

There are several models for representing concepts that can be learned by classifiers. We will study three in this course: decision trees, perceptrons, and feed-forward neural networks. There are many other models in the literature such as decision lists, forests, support vector machines, nearest neighbor and certain types of regression. Each model has strengths and weakness and certain types of concepts they work well for. Many of the underlying principles we cover here are shared among all models, though.

6.2 Decision Trees

The decision tree model is familiar to most people even outside of computer science (see figure)[1]. It is a well-known way to formulate a decision process for determining an outcome based on certain variables. The variables themselves are the features and the outcome is the classification.

For example, perhaps you enjoy playing tennis during the year and on any given day you use the weather to determine if you will play tennis. You many never play if it is raining or extremely hot or windy. But if the temperature is mild or slightly warm, you may decide to play as long as the humidity is low. When it is cool you may play even if the humidity is high. You could organize your decision process into a tree consisting of vertices and labeled directed edges. Each interior node corresponds to a question you ask (a feature of the dataset) and each edge corresponds to an answer (a valid value for the

[1] image source: *http://www.cse.unsw.edu.au/~billw/cs9414/notes/ml/06prop/id3/id3.html*

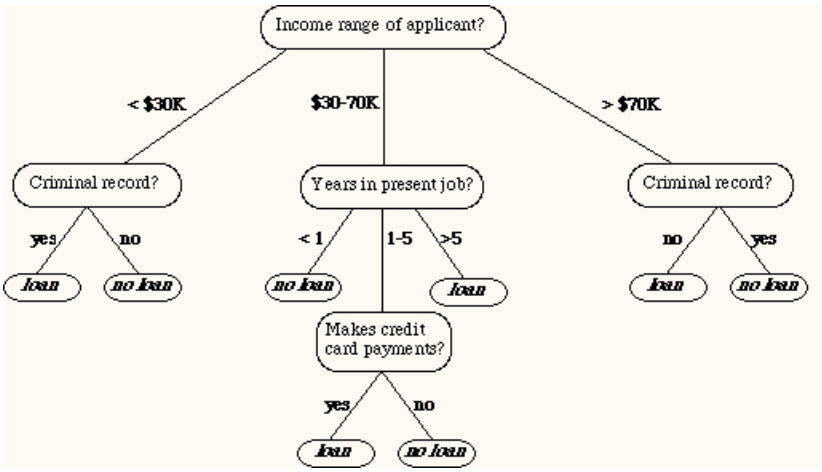

Figure 6.1: A decision tree for approving loans

feature). The leaves of the decision tree are categories corresponding to your decision: Yes or No.

A decision tree learning algorithm, if given access to a set of classified observations consisting of weather observations and your decision about playing tennis, would be able to construct a tree that accurately determined your decision, based solely on the feature vectors of weather data. How *good* this tree is would not really be known until it was tested on some unseen data, so the error rate on a test set could be determined. It the rate is low, we may conclude the tree has high prediction value.

You should convince yourself that for any consistent training set, a decision tree can always be created that has a zero percent error rate on the training set. (In the most extreme case, every single observation can correspond to a unique leaf and every outcome is therefore determined by a single dedicated path in the tree). Such a tree would be disastrously complex, and most likely useless on any feature vector not in the training set. Succesful earning algorithms employ *Occam's Razor*: the simpler solution is usually preferable. A large tree

with 0% error on the training set may have 50% error on a test set. But a small, simpler tree might achieve 5% error on both. Since we can't know the test set performance beforehand, we'll use the rule of thumb that favors small trees over large ones.

Small trees may not always be supported by the data. For example your decision to play tennis could be purely random, or determined by a factor not in the tree, such as the time of day or the availability of your tennis partner. In this case we would probably conclude that the target concept is not learnable by our model, at least based on the given feature set.

Exercises

The WillWait? data set is taken from Russel and Norvig's text and is a standard data set in learning decision trees. The data set (see Figure 6.2) consists of 12 observations, 10 categorical features each and one binary classification: whether to wait at a restaurant or not. Some of the features are binary, others have 3-4 possible values.

(11) Construct, by hand, a small decision tree that correctly classifies all 12 observations in the dataset. The size of your tree will be defined simply as the number of nodes in the tree, including the root and all leaves.

(12) How much smaller can you make your tree if you allow some classification error in the training set? Is the amount of error acceptable? Do you think this smaller tree will generalize to unseen data as well as the larger tree? Or better?

(13) Sketch out a tree that classifies every example using a terrible algorithm – create a new leaf for each observation. Don't try to optimize anything. Complete enough of the tree to convince yourself this algorithm always works.

Example	Attributes										Target
	Alt	*Bar*	*Fri*	*Hun*	*Pat*	*Price*	*Rain*	*Res*	*Type*	*Est*	*Wait*
X_1	T	F	F	T	Some	$$$	F	T	French	0–10	T
X_2	T	F	F	T	Full	$	F	F	Thai	30–60	F
X_3	F	T	F	F	Some	$	F	F	Burger	0–10	T
X_4	T	F	T	T	Full	$	F	F	Thai	10–30	T
X_5	T	F	T	F	Full	$$$	F	T	French	>60	F
X_6	F	T	F	T	Some	$$	T	T	Italian	0–10	T
X_7	F	T	F	F	None	$	T	F	Burger	0–10	F
X_8	F	F	F	T	Some	$$	T	T	Thai	0–10	T
X_9	F	T	T	F	Full	$	T	F	Burger	>60	F
X_{10}	T	T	T	T	Full	$$$	F	T	Italian	10–30	F
X_{11}	F	F	F	F	None	$	F	F	Thai	0–10	F
X_{12}	T	T	T	T	Full	$	F	F	Burger	30–60	T

Figure 6.2: The WillWait Dataset

(14) How many nodes will a tree constructed like the previous problem contain in the worst case? Let n be the number of observations and m be the number of features.

6.3 Measuring the value of knowledge

Measuring Information

Alex and Betty want to exchange messages using an alphabet of 8 characters. They will encode their messages in binary. How many binary bits are required to encode each letter? Clearly 3 will suffice because $2^3 = 8$ and the 8 different 3-digit binary strings can each stand for a different letter. A=000, B=001, etc.

Is it possible to use fewer than 3 bits per letter? At first glance the answer may obviously seem to be "no." But imagine an extreme case where all of the words in the language only used two letters: A and H. Then they could use one bit: A = 0, H = 1 and communicate perfectly well. Even though there are 8 characters, they are not all equally prevalent in the language. If we use frequency analysis of

words in the language and determine some letters are more common than others, we can possibly get by with fewer than 3 bits.

Assume that we determine that A is the most frequent letter, occurring 50% of the time, while B,C,D,E,F,G,H each occur $\frac{1}{14}$ of the time, making up the other half. Let's devise a scheme that uses *on average* fewer than 3 bits per character. By "on average" we mean that if we encode a message of n letters, it will use fewer than $3n$ bits, assuming the letters in the message occur with the frequencies defined above. (More precisely, we're computing the *expected number of bits* in a letter – to be discussed below).

Since A is the most common, we could try encoding it with only 1 bit, say "0". The remaining 7 letters can be distinguished using only 3 bits each: B = 001, C = 010, etc. This would reduce the average bits-per-letter. But there's a problem: messages are ambiguous. How would you decode 00100010? is it ACBA, or BABA? Instead we'll use four bits for B-H, with high bit 1: B = 1000, C = 1001, etc. Now when decoding, a "0" signifies an A while a "1" signifies the start of a 4-bit code. 0010001001 = 0 0 1000 1001 = AABC.

The average bits-per-character, according to the probabilities given, is

$$B_{avg} = \frac{1}{2}(1) + 7 \times \frac{1}{14}(4) = \frac{5}{2} = 2.5.$$

We can interpret this figure, 2.5 bits, as a consequence of the skewed probability distribution of the 8 letters. If letters were equally probable then fully 3 bits are required to communicate each letter. This is a type of worst-case scenario. There is no reason to prefer any letter over any other. There is no advantage to knowing the probability distribution. Every single one of the 3 bits is informative. On the other extreme, a case where only letter "A" is ever used is a best-case scenario. Here 0 bits would be required. There is no information conveyed by knowing the next letter is an "A", because by knowing the distribution, that fact is obvious.

The 2.5 bit case is a middle ground. There is a certain advantage to knowing the distribution beforehand: you can guess the letter will

be A and half the time you'll be right. The leading bit "0" confirms your guess. If the letter is not "A", the leading bit "1" tells you that you need to see 3 more bits to determine the letter. On average there's 2.5 bits of "information" conveyed by knowing the next letter.

This idea of information is formalized in the next section on *entropy*.

Information and Entropy

Colloquially, the "entropy" of a system can be described as the amount of disorder or disarray in the system. Entropy and information are closely related. Consider for example a library. If every book is ordered on the shelves by author and title, then relatively little information is required to describe the arrangement of books. A list of titles and authors may suffice, perhaps along with the position of the shelves. But if somebody comes in and dislodges each book, throwing them all over the various floors of the library, then the system is suddenly very hard to describe. Each individual book will be in a specific position, and orientation, possibly open to a certain page. Some pages might be bent or ripped. Books may be intertwined with each other, opened partially, or bent fully back. The high-entropy arrangement of books is full of information.

These concepts can be applied to probability distributions, as motivated in the previous section. A uniform distribution, where each of n events is equally likely, is a high-entropy, high-information situation because any event could occur with the same probability. Conveying which event occurs takes $\log_2(n)$ bits. On the other hand, a distribution over n elements where only one element has all the probability is low-entropy, low-information. The outcome is already known. 0 bits are required to convey the event, because it's always the same. (And it just so happens that $\log_2(1) = 0$, so a bit of a pattern emerges).

We'll define the *information* of an event with probability p as simply $I(p) = -\log_2(p)$. For example an event that has probability $\frac{1}{8}$ will have an information value of $I(\frac{1}{8}) = -\log_2(\frac{1}{8}) = 3$, in agree-

ment with the previous example. An event with probability 1 has information value 0. An event with probability 2^{-20} has information content 20. Rare events have more information, because they are less likely, and more "news-worthy" when they occur.

The the *Shannon entropy* (or just *entropy*, attributed to Claude Shannon) is defined as the *expected value of the information* of a probability distribution.

$$h(p_1, p_2, \ldots, p_n) = -\sum_{i=1}^{n} p_i \log_2(p_i)$$

(Recall that expected value of a random variable is the sum of the values taken by that random variable, each weighted with its respective probability. For example, if a fair die is rolled, the expected value of the square of the value displayed on the die is $\frac{1}{6}(1+4+9+16+25+36) = \frac{91}{6}$)

Returning to the 8-character alphabet example, the Shannon entropy of the uniform distribution is

$$h(\frac{1}{8}, \frac{1}{8}, \ldots, \frac{1}{8}) = -8(\frac{1}{8}\log_2(\frac{1}{8})) = 3$$

while the skewed distribution is

$$h(\frac{1}{2}, \frac{1}{14}, \ldots, \frac{1}{14}) = -(\frac{1}{2}\log_2(\frac{1}{2}) + \frac{7}{14}\log_2(\frac{1}{14}) = 2.403$$

The first value agrees with the 3 bits required. The second is slightly less than 2.5, suggesting that perhaps a more efficient coding scheme can be found, if these concepts of entropy and information are truly related.

In fact they are very related. If we accept the axiom that an event of probability p takes $-\log_2(p)$ bits to communicate to someone when it occurs, then the definition of entropy given above is exactly equivalent to the definition of *expected number of bits* in a letter given above. Of course the number of bits actually transmitted

depends on the chosen encoding scheme, so the equivalence seems to only be valid in the optimal (or perhaps limiting) case. The Shannon entropy provides a lower bound on the number of bits required in any such encoding scheme and, sometimes, that bound is achievable.[2]

Exercises

(15) What is the entropy of the values in the Wait? column of the WillWait dataset? We will refer to the entropy of the classifications as simply the entropy of the dataset.

(16) If you know that it is Friday and define a new dataset based on only the relative rows, what is the entropy of that dataset? What is the entropy if it is *not* Friday?

(17) Compute an "average entropy" for the value *Friday* by averaging the two entropys corresponding to *Friday*=*T* and *Friday*=*F* that you found in th previous exercise. Be sure to weight each probability with its frequency (7/12 of the observations happen when Friday=F). How much, on average, does the entropy decrease from the original dataset if you know the value of "Friday"?

(18) Find a value for a feature in which the resulting entropy is 0. Find a value for a feature in which the resulting entropy is 1. Find a feature where every value has an entropy of 1.

6.4 Decision Tree Learning

All this talk about information and entropy will be put to use now in our decision tree learning algorithm. When any decision tree is

[2] This is the essential content of Shannon's source coding theorem. Two schemes, Huffman codes and arithmetic codes, can be shown to be optimally close to the Shannon bound under certain assumptions.

used to classify a previously unseen observation, the initial entropy h_0 can be defined as the entropy of the probability distribution over all the outcomes in the set. As the observation is sifted down through the tree, the distribution over outcomes changes until eventually a leaf is reached and the final entropy h_f is necessarily 0, because the sole outcome consistent with the observed features, is known. Since $h_0 > 0$ for all but trivial problems, the entropy can be assumed to decrease throughout the path from the root to the leaf of the tree. A well-constructed tree will make this path, on average, as short as possible and a reasonable way to do that is to maximize the entropy loss at each step. Conversely, this can be stated as increasing the information gain at each step. So, our job in making a decision tree is to form nodes out of questions whose answers provide the most information.

We'll work through an example with the Play_Tennis dataset.

Table 6.1: The Play_Tennis dataset

ID	Outlook	Temp	Humidity	Windy	Play?
1	sunny	hot	high	false	no
2	sunny	hot	high	true	no
3	overcast	hot	high	false	yes
4	rainy	mild	high	false	yes
5	rainy	cool	normal	false	yes
6	rainy	cool	normal	true	no
7	overcast	cool	normal	true	yes
8	sunny	mild	high	false	no
9	sunny	cool	normal	false	yes
10	rainy	mild	normal	false	yes
11	sunny	mild	normal	true	yes
12	overcast	mild	high	true	yes
13	overcast	hot	normal	false	yes
14	rainy	mild	high	true	no

This dataset D has 14 rows, each containing a 4-element feature vector and a binary classification: yes or no (the IDs are not features.) These 14 rows contain 9 'yes' classifications and 5 'no' classifications. We'll define the entropy of D as the entropy of the frequency distribution of its classifications. So $h(D) = h(\frac{5}{14}, \frac{9}{14}) = 0.9403$. Since the distribution is slightly skewed, the classification does not contain 1 full bit of information. But it's pretty close.

Now we define a refinement $D|_{f=v}$ of a dataset D as a subset of the rows of D in which the feature f has value v. For example $D|_{temp=hot}$ is the set of rows labeled {1,2,3,13}. The *entropy loss* of a refinement of D is defined as the difference in the entropies of the two datasets:

$$L(D|_{f=v}) = h(D) - h(D|_{f=v})$$

The dataset $D' = D|_{temp=hot}$ has an entropy of $h(\frac{2}{4}, \frac{2}{4}) = 1$ so the entropy loss of D' is $L_h = -0.0597$. (A negative entropy loss is not a good sign – this piece of information actually gives you less of an idea of the right classification than no information at all!). You should verify that $D|_{temp=mild} = h(\frac{4}{6}, \frac{2}{6}) = 0.9183$ and $D|_{temp=cool} = h(\frac{3}{4}, \frac{1}{4}) = 0.8113$ corresponding to entropy losses of $L_m = 0.0220$ and $L_c = 0.1290$.

Now we have some assumptions to state. In particular, we're assuming this dataset is representative of the underlying concept we're hoping to learn and that the frequencies observed here are indicative of the frequencies we will encounter in the yet-unseen test dataset. So if $\frac{2}{7}$ of the days are cool in the dataset, we assume about $\frac{2}{7}$ of the days will be cool in the test set. Under this fairly plausible assumption we can define the *expected entropy loss* of the feature "Temp".

Given any observation at random, the expected entropy loss of the feature "Temp" is $L(temp) = \frac{4}{14}L_h + \frac{6}{14}L_m + \frac{4}{14}L_c = 0.0292$. That is to say, of the 0.9403 bits of information needed to classify the members of this dataset, the feature "Temp" conveys, on average, 0.0292 bits of information. Is that good? Well it all depends on the other features. Maybe one of the other 3 is better.

Once the expected entropy loss for each feature is computed, the feature with the maximum entropy loss is selected as the root node of the decision tree, because this node is expected to give the most information quickly. This idea the extends immediately to each refinement of the dataset. Assume, for example, that Temp is the best feature. We would make "Temp" the root of the decision tree and add three edges: hot, mild and cool. Each edge defines a refinement of D. Recursively apply the procedure to find the best feature on each of these three subsets of D (omitting of course "Temp"). The procedure continues until the current refinement has an entropy of 0, meaning all elements are in the same class.

Learning Decision Tree Pseudocode

A recursive implementation of Decision-Tree-Learn is given below. The function returns a node representing the root of the decision tree. The decision tree is stored as a directed graph with labeled nodes and edges. Nodes are labeled with features in the dataset, while edges are labeled with values corresponding to that feature. In this way, nodes act as questions and edges act as answers.

While this code is intended to be fairly implementation-independent, it is assumed that *node* is an object capable of maintaining a list of children, as in any standard tree data structure. These children are stored as a list of (*key, value*) pairs where the *key* is the edge label and the *value* is the label of the child node.

This code requires a nuber of helper function to work. The code refers to *find-best-feature*, *entropy* and *refine*. Additionally you need easy ways to determine all the features in a dataset, all the values in a feature, and the frequency distribution of classes of a dataset.

Once the tree has been computed, it will be handy to have a quick way to visualize the tree itself. Although an actual graphical sketch might be nice, it's only really feasible for small trees. An indented list or outline form is more practical, and easy to implement. For example, an outline of tree in Figure (???) is

Algorithm 10 Decision tree learning algorithm

function DECISION-TREE-LEARN(D)
    input: D, a non-empty list of ($\vec{x}$, y) where $f(\vec{x}) = y$.
    if entropy(D) == 0 **then**
        $class = D[0, 1]$ ▷ the class of the first element in D
        $leaf$ = node($class$, type='leaf')
        return leaf
    f = find-best-feature(D)
    node = node(f, type='internal')
    for each $value$ in feature f **do**
        D' = refine($D, f, value$)
        if D' is empty **then**
            continue
        else
            child = Decision-Tree-Learn(D')
            add $\{value : child\}$ to $node.children$
    return $node$

Algorithm 11 Find the feature in a dataset with maximum entropy loss

function FIND-BEST-FEATURE(D)
    $h_0 = entropy(D)$
    for each feature f in D **do**
        $h_f = 0$
        for each value v in f **do**
            $D' = restrict(D, f, v)$
            $p_v = len(D')/len(D)$
            $h_v = entropy(D')$
            $h_f = h_f + p_v \cdot h_v$
        $L_f = h_0 - h_f$
    return f for which L_f is maximum

Algorithm 12 Find the entropy of a dataset

```
1: function ENTROPY(D)
2:     classes = set of classifications in D
3:     h = 0
4:     for each c in classes do
5:         f_c = number of occurrences of c in D
6:         p_c = f_c / len(D)
7:         if p_c > 0 then
8:             h = h + p_c · log_2(p_c)
9:     return −h
```

```
* Income Range?
  * < 30K
    * Criminal Record?
      * Yes --> Loan
      * No --> No Loan
  * 30-70K
    * Years in present job?
      * < 1 --> No loan
      * 1-5
        * Makes credit card payments?
      * Yes --> Loan
      * No --> No Loan
      * > 5 --> Loan
  * > 70K
    * Criminal record?
      * No --> Loan
      * Yes --> No Loan
```

Exercises

(19) Implement functions to compute entropy of (a) a list of probabilities (b) a list of frequencies and (c) a 2-D array of values representing a data set, where each row is an observation and

the final column contains the classification.

(20) Manually calculate the best feature for the PlayTennis data set. You may want to use the entropy functions in the previous exercise.

(21) The choice of data structures to represent a data set and a decision tree will have a impact on the complexity of your code. Brainstorm at least 5 data structures for a data set and 3 for the decision tree. Compare the complexity of the necessary functions on each. (Time complexity is important here, but focus first on programming complexity. Slow and right is better than fast and wrong.) Consider lists, arrays, nested lists, tuples, dictionaries, sets, hashes and custom data structures that combine all of the above.

6.5 Coding Project: Play Tennis?

You will implement the algorithms presented in this section and build a decision tree to correctly classify the observations in the PlayTennis dataset.

The Problem

Play-Tennis is a well-known beginning data set in machine learning literature. It first appeared in (Quinlan 89?). The fourteen observations consist of 4 categorical features and one binary classification. While there are many decision trees that correctly classify all the observations, there is only one tree that uses entropy-loss at each step to determine the best feature for splitting the dataset.

Design specifications

You will implement the algorithms described above and any helper functions needed to produce a data structure representing the decision tree learned by Decision-Tree-Learn. You should read file

play-tennis.csv, parse it into an appropriate representation, learn the decision tree, print the decision tree in outline form, and finally test the decision tree to verify all 14 observations are correctly classified.

You should also write functions that can produce the following statistics about your tree: total number of nodes (including leaves), number of leaves, number of non-leaves, average path length (for each leaf, count the number of nodes from the root to the leaf, inclusive, and average these counts). Print this data along with the tree outline.

Your tree outline should be formatted like the sample above. You should additionally include with each node, the entropy of the dataset associated with that node. For example, the root node would display the entropy of the entire dataset while leaf nodes would have an entropy of 0.

Implementation Notes

As discussed in the exercises, finding the right data structures will be crucial for this problem. For the data set you should be able to easily refine the dataset, extract the values in any feature, and extract a subset of rows. The original .csv file does not include ID numbers on the data, but you may find it very helpful to add unique IDs so you can represent observations with just an index instead of the entire row. Beware of nested-list type structures in Python, especially if you try copying. Remember if you're dealing with an object or a pointer to an object. If at all possible, avoid copying entirely. It is not necessary in this code, creates time and space overhead, and can lead to subtle errors.

The tree itself can be modeled after any standard tree or graph data structure. As suggested in the pseudo-code, you will need to make sure you can label nodes and edges and distinguish leaf nodes from internal nodes. The list of children of a node can be easily handled as a list of tuples or a dictionary of (edge-label: node-label) pairs.

Listing 2 Using PyGraphViz

```
import pygraphviz as pgv

G = pgv.AGraph(directed=True)
G.add_node(1, label="Age?")
G.add_node(2, label="Accept")
G.add_node(3, label="Reject")
G.add_edge(1,2, label = ">25")
G.add_edge(1,3, label="<= 25")
G.layout(prog="dot")
G.draw("minimal-tree.png")
```

Extensions

GraphViz[3] is open-source graph visualization software. It consists of a language for specifying graphs and routines that can draw these graphs as graphics files. Python has an interface to GraphViz called pygraphviz that makes it easy to specify graphs in pure python and generate graphs from them.

For this extension, learn how to build and draw graphs in Python using pygraphviz and add the ability to your code to draw a decision tree's graph. You will need to label the nodes and edges and may, therefore, need to specify a way to abbreviate labels. Otherwise the tree may become unwieldy.

A sample of a tree produced by pygraphviz (based on data in the next programming project) is shown in Figure 6.3

To help you get started, see Listing 2 for sample code to produce a minimal example of a decision tree using pygraphviz.

6.6 Performance Metrics

It is always possible to build a decision tree with a 0% error rate on a consistent data set. (Consistent means that if $x_1 = x_2$ then

[3] http://graphviz.org

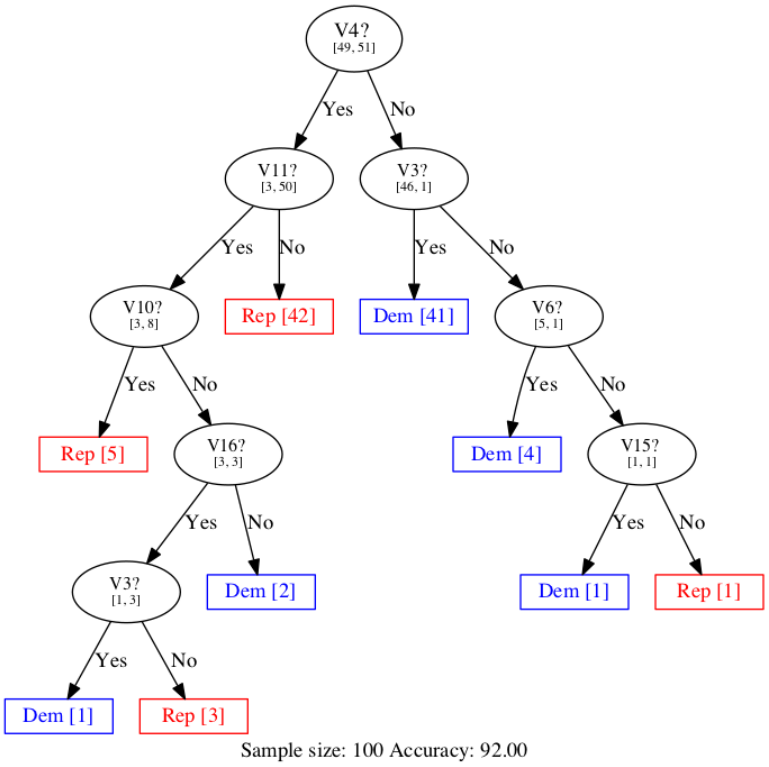

Figure 6.3: A Decision Tree drawn by GraphViz

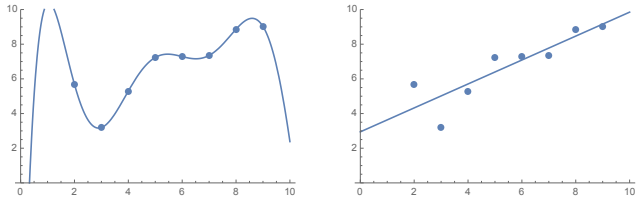

Figure 6.4: High and low degree polynomial models

$f(x_1) = f(x_2)$.) But this tree could be very complicated and may not capture an underlying structure that generalizes well. To tell the applicability of a learning model beyond the data set it was trained on, we will use a second data set – the *test set* to validate the accuracy of the model. If the mdoel has truly caputred an underlying concept (it closely models the unknown f), then it should have a low error rate on the test set as well.

A similar phenomenon occurs with polynomial regression. A set of n points in the plane can always be perfectly interpolated with a degree $n - 1$ polynomial. It is very unlikely, though, that this polynomial will have any applicability beyond the n original points. A low-degree polynomial model is almost certainly a better choice. (See Figure 6.4.)

When a learning model matches the training data too closely that it performs poorly on the test data, we say that model has *overfit* the data. In decision tree learning, a helpful technique is to analyze the test set error rate of the tree learned as a function of the training set size. Typically the error rate will decrease until a certain point and start increasing again. This optimum *point of no return* is a good indicator of the onset of overfitting. Such a graph is called a learning curve (see Figure 6.5)

When analyzing data in this way it is critical to keep the training set and test set separate and never unduly allow the test set to change your choice of model. The test set can be used as a benchmark of accuracy but should never be used to help train the tree. Otherwise

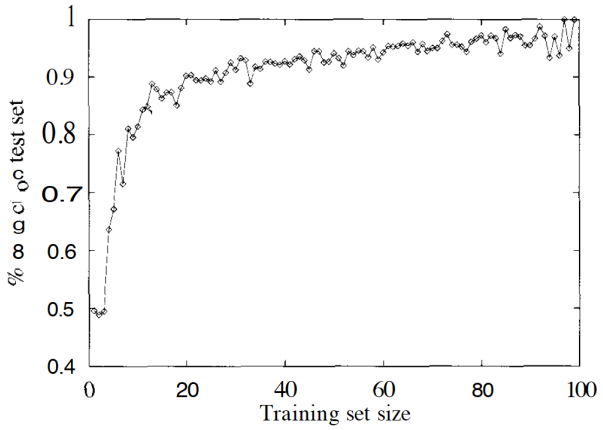

Figure 6.5: A Learning Curve (placeholder)

it loses it purity as an unseen test set. Given an initial dataset it is customary to partition it so that 50-90% of the data is used for the training and the remainder is used for the test set. If the learning algorithm is run multiple times in succession, a new training/test set partition should be randomly chosen so the results will not be conditioned on the model's performance against one fixed test set.

6.7 Additional Considerations

Inconsistent data sets

So far we have assumed consistency of the training set. Otherwise it will be impossible to achieve a 0% error rate on the training set. Worse than that, the algorithm given above will fail if a set of identical observations are found with a non-zero entropy.

One simple fix is to modify the algorithm so that if the current dataset cannot be split, because all features have the same values for all items in the set, but the classifications are not all identical, then

return a leaf node with a label equal to the majority classification in the dataset.

Missing Values

Real-world data sets rarely contain entirely usable data. A frequent issue is that some observations may have missing values for one or more features. You may be tempted to just throw these observations out, but they often still contain useful information, and more often than not you won't have enough rows to learn a meaningful hypothesis from if you demand each observation be fully populated.

Missing values can affect the training set and the test set. Different techniques can handle each case. With missing training data, you can preprocess the data set and fill in the missing values with reasonable guesses. Assume that observation n is missing the value of feature f. One way you can fill in the value is to replace it with the majority value of feature f across the entire dataset (*i.e.* the most common value). If the feature is continuous valued then instead of the majority you can replace it with the average or median value of the feature. An alternative method to each of these is to compute the majority (or mean or median) relative to only the subset of the dataset that has the same classification as row n.

Missing data in the test set can be handled while the resulting tree is being traversed. For one, it may be possible to classify the unseen observation without referring to the missing feature – the tree may classify it using only the other rows. In the case where the missing feature appears while traversing the tree, the algorithm can take *all* paths out of the node corresponding to the unknown feature and tabulate the *majority* classification. **Give an example**

Continuous Valued Features

The datasets in this unit only contain categorical features. But real-world datasets often contain quantitative values. For example a patient's medical records will contain temperature, blood pressure and

pulse rate data, along with a diagnosis. In order for a decision tree to deal with temperature, it will be necessary to find a *split-point* to convert the continuous data into categorical. For example $temp > 101$ would be a binary feature resulting from the original temperature data.

Though in some cases the split point can be determined before hand (for example it may make sense to split *age* at 18, 21, or 25 in some contexts), modern decision tree learning algorithms determine the best split point dynamically as the tree is being built. A simplistic way to do this is to consider every possible split point and choose the one with the most information content. The requires doing an entropy computation for each value in the feature – *not* every observation in the dataset. A medical record database of 10,000 patients will probably contain at most 70 different temperatures (98.0-105.0 in increments of 0.1). Even so, there are better approaches than this brute-force technique.

Pruning

As we have seen, large decision trees are prone to overfitting. Trees typically grow in size as the size of the training set increases, yet limiting ourselves to a small training set runs the risk of choosing an unrepresentative sample. *Pruning* is a technique that allows us to process a large training set but avoid overgrowth of the tree by adding nodes only when they are statistically relevant. Is it possible to prune while the tree is being built, or after.

To prune the tree during the learning phase you need a procedure to determine if the current node should be split or if it should become a leaf. For example if 50 observations are classified "Play Tennis" and 3 observations are classified "Do Not Play Tennis", it might not be worth splitting the dataset to deal with just 3 examples. They may be spurious or just arise from random noise in the dataset and not part of a true underlying concept based on these features (for example, maybe one beautiful cool day you didn't play tennis because you had the flu, and another day you didn't play because your car had a flat

tire.) An easy heuristic way to determine whether to split is to set a minimum entropy slightly above 0 as the test for creating a leaf and then to label that leaf with the majority classification.

Another more justifiable technique, called *Chi-Squared Pruning* uses statistical testing to determine if the current feature is likely to provide additional information or not, based on the frequency distribution of classifications in the current dataset and its parent dataset. If the feature is unlikely to be informative, the split will not occur and a majority-labeled leaf is created.

Pruning after-the fact can be done by successively removing subtrees from the original tree and comparing the test error rate. If a subtree can be removed and the test accuracy is not significantly harmed, then that subtree can probably be safely pruned from the tree.

6.8 Programming Project: Party Affiliation

You will build a decision tree from a dataset of congressional voting record. It contains about 500 rows and 16 features. Based on voting records you will be able to predict party affiliation of the congress members

The Problem

This publicly available dataset (downloaded from here) contains voting records for 16 different congressional bills and the party affiliation (democrat or reupblican) as the classification. The data is *dirty* because some of the votes are missing. Features are binary variables (y or n) and each row is labeled with a unique ID.

In the first part of this project you will simply build a tree using your existing Decision-Tree-Learn algorithm, essentially ignoring missing data. You will produce some graphs analyzing the error rate of your algorithm on the training and test sets as a function of the training set size. Then you will enhance your algorithm by dealing

with missing data in different ways.

Finally you will investigate some techniques for pruning the tree you learned and make some decisions about the *best* tree for the concept.

Specification

Implementation Notes

Extensions