CS181 Lecture 7 — Overfitting

Today we continue our discussion of the decision tree learning algorithm. The main focus will be on the phenomenon of overfitting, which is an issue for virtually all machine learning algorithms. After discussing why overfitting happens, we will look at several methods for dealing with it.

1 Comments on ID3

1.1 Information Gain Heuristic

Several comments on the ID3 algorithm are in order. First, let us try to understand better what kinds of splits the information gain criterion selects. Recall the definition of information gain. It begins with the definition of Entropy, which is a measure of the disorder in the data $\mathbf{D}$:

$$\text{Entropy}(\mathbf{D}) \quad = \quad -\sum_{c \in C} \frac{N_c}{N} \log \frac{N_c}{N} \tag{1}$$

Suppose the we are considering splitting on attribute X with current dataset D. For a value x of X, let $\mathbf{D}_x$ be the subset of $\mathbf{D}$ such that $X = x$. We can characterize the amount of disorder remaining after splitting on X by the weighted average of the entropies of the $\mathbf{D}_x$:

$$\text{Remainder}(X, \mathbf{D}) \quad = \quad \sum_x \frac{N_x}{N} \text{Entropy}(\mathbf{D}_x) \tag{2}$$

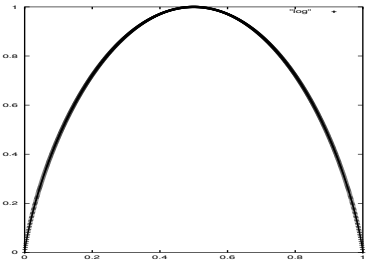
We then characterize the improvement or gain from splitting on X by the difference between the original entropy and this remainder:

$$\text{Gain}(X, \mathbf{D}) \quad = \quad \text{Entropy}(\mathbf{D}) - \text{Remainder}(X, \mathbf{D}) \tag{3}$$

ID3 chooses the attribute that maximizes this measure, called the *information gain* of the attribute.

To understand this heuristic better, note first that since Entropy($\mathbf{D}$) is fixed, the attribute that maximizes Gain is the one that minimizes Remainder. The Remainder itself is made up of two components. Let us examine each in turn.

One of the components of Remainder is the entropy of the data sets D_x for the subpartitions. Recall the shape of the entropy function for binary data:



Here, $f(x) = x \log x + (1-x)\log(1-x)$ is a concave function. Its derivative is $\frac{1}{\ln 2}(\ln x - \ln(1-x))$, which is ∞ at 0, and $-\infty$ at 1. This means that the greatest difference in entropy happens near the extremes of 0 and 1, while the difference in the entropy of mid-range distributions is relatively small. There is not much difference between mediocre partitions. For example, the entropy of the 5 :: 9 partition is 0.94, while that of 7 :: 7 is 1. On the other hand, there is a big difference between partitions that look similar at the extremes. For example, the 1 :: 6 partition has entropy 0.59, while the 0 :: 7 partition has entropy 0. This implies that the Remainder function selects strongly for extreme partitions in some of the subpartitions, even if the other subpartitions are very mixed. Another way of putting it is that Remainder prefers one extreme and one very mixed subpartition to two fairly well-sorted subpartitions.

The second component of the Remainder function is the weights $\frac{N_x}{N}$. These mean that getting low entropy for large subpartitions is more important than for small subpartitions. To summarize, then, the information gain criterion will try choose an attribute to split on that classifies large subpartitions extremely well. Let's have a look at how this works in an example. The original data set is as follows:

Skin	*Color*	*Thorny*	*Flowering*	Class
smooth	*pink*	*false*	*true*	*Nutritious*
smooth	*pink*	*false*	*false*	*Nutritious*
scaly	*pink*	*false*	*true*	*Poisonous*
rough	*purple*	*false*	*true*	*Poisonous*
rough	*orange*	*true*	*true*	*Poisonous*
scaly	*orange*	*true*	*false*	*Poisonous*
smooth	*purple*	*false*	*true*	*Nutritious*
smooth	*orange*	*true*	*true*	*Poisonous*
rough	*purple*	*true*	*true*	*Poisonous*
smooth	*purple*	*true*	*false*	*Poisonous*
scaly	*purple*	*false*	*false*	*Poisonous*
scaly	*pink*	*true*	*true*	*Poisonous*
rough	*purple*	*false*	*false*	*Nutritious*
rough	*orange*	*true*	*false*	*Nutritious*

Overall, there are 5 *Nutritious* cases and 9 *Poisonous* cases for an entropy of 0.94. Splitting on the various attributes results in the following subpartitions:

Attribute	Subpartitions				Remainder	Gain
Skin	Value	*Nutritious*	*Poisonous*	Entropy	$5/14 * 0.971$ $+ \quad 5/14 * 0.971 \quad = 0.694$ $+ \quad 4/14 * 0$	0.247
	smooth	3	2	0.971		
	rough	2	3	0.971		
	scaly	0	4	0		
Color	Value	*Nutritious*	*Poisonous*	Entropy	$4/14 * 1$ $+ \quad 6/14 * 0.918 \quad = 0.911$ $+ \quad 4/14 * 0.811$	0.029
	pink	2	2	1		
	purple	2	4	0.918		
	orange	1	3	0.811		
Thorny	Value	*Nutritious*	*Poisonous*	Entropy	$7/14 * 0.986$ $+ \quad 7/14 * 0.592 \quad = 0.788$	0.152
	false	4	3	0.986		
	true	1	6	0.592		
Flowering	Value	*Nutritious*	*Poisonous*	Entropy	$6/14 * 1$ $+ \quad 8/14 * 0.811 \quad = 0.892$	0.048
	false	3	3	1		
	true	2	6	0.811		

We see that *Skin* produces by far the best information gain, because it has one perfectly classified subpartition, even though the other two subpartitions are even worse than the original distribution.

1.2 Greediness

The next comment on ID3 is that it is a greedy algorithm. It uses information gain to greedily decide what attribute to split on next, and never questions its decision. Choosing the node that has the highest information gain does not necessarily lead to the shortest tree. For example, suppose there are 100 features, and the true concept is $X_1 = X_2$, i.e., the classification is *true* if X_1 and X_2 are equal. The remaining 98 features are irrelevant. Here, considered individually, X_1 and X_2 are both useless. It is only in conjunction with the other that they are useful, but in that case they entirely determine the class. Since ID3 only considers one feature at a time, it is no more likely to split on X_1 or X_2 as it is to split on any one of the other features. Once it has split on X_1 or X_2, it will of course split on the other at the next step, but it may take a long time until it splits on X_1 or X_2. So ID3 will likely grow a much larger tree than necessary. Cases like this, in which two attributes are not individually predictive but work as *co-predictors*, would not be a problem for an algorithm that looked ahead one step in deciding what to split on.

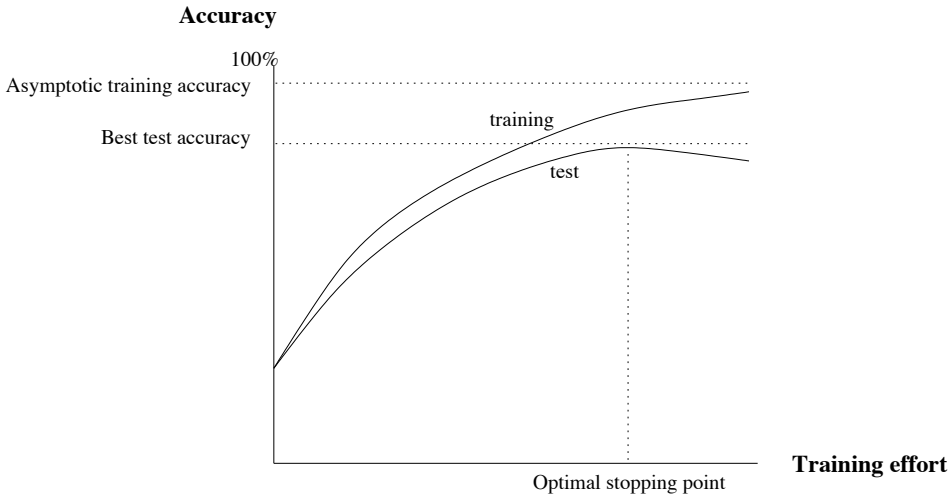
2 The Overfitting Phenomenon

How well does ID3 actually perform as a learning algorithm? Given a consistent training set (where no two cases have the same attributes but different class),

ID3 will learn a tree that scores perfectly on the training set, unless it happens at some point that no attribute has positive information gain (why?). Furthermore, for an inconsistent training set, it will learn a tree that has the fewest possible errors on the training set, unless it happens at some point that no attribute has positive information gain (again, why?).

So, it seems like ID3 is a great algorithm, right? True, it always gets a hypothesis with as low an error on the training set as possible. But that does not necessarily mean that it learns a classifier that will generalize well. In fact, getting a perfect score on the training set might not be the best thing to do! Doing better on the training set may actually lead to doing worse on a test set. This phenomenon is called overfitting.

We can see the effect by a graph that tracks the performance of the different trees grown during ID3's process of growing a decision tree. The x-axis shows the number of nodes in the tree, while the y-axis shows the percentage of correct classifications on the training and test sets. The following graph is typical: the training set performance continues to improve, while the test set performance peaks before the complete tree has been grown and starts to dip afterwards.



Why does this happen? The problem is that the full decision tree is fitting the noise in the training data. The problem is that near the bottom of the tree, there are very few cases on which to base decisions, so the algorithm may make a splitting decision that is not statistically supported by the data. Such an unsupported decision will override statistically supported decisions at a higher level. It may be the case that the higher level decision was in fact correct, but the lower level one is not.

Overfitting is a huge issue in machine learning. It comes up for just about all learning algorithms. The basic issue is that there is always a tradeoff between producing a model that fits the data as well as possible, and a model that

generalizes well.

3 Experimental Methodology

One moral of overfitting is that we need to be very careful about our experimental methodology in testing machine learning algorithms. Early machine learning work reported success by showing that algorithms could learn a model that fit the data. As we know, this is the wrong goal. We want algorithms that generalize well to unseen data. It is vital to test algorithms not on the training data but on test data.

This implies that when we run machine learning experiments, we need to set aside some data for testing. We also want to run several experiments, so that we can characterize the performance of an algorithm on several different training and test sets, to get statistically better supported results.

This is fine if we have plenty of data to work with. We just divide it up into 10 different sets, and divide each set up into training and test sets. But in the real world data is often hard to come by, and we simply can't afford to divide it up into small chunks.

A way to deal with this issue is to use the *cross-validation* method of performing experiments. This method works as follows. The data is divided up into k equal sized *folds*. A typical value of k is 10. Then, k different experiments are run. In each experiment, $k-1$ folds are used as training data, while the remaining fold is used as test data. Over the course of the k experiments, each fold gets used as test data exactly once. This method allows multiple experiments to be run on a data set that is only big enough for one training and test set. Although the experiments are not completely independent of each other, since they have overlapping training sets, running cross-validated experiments provides more significant results than running a single experiment, and the method is completely standard in the field.

A special form of cross-validation is called *leave-one-out* cross-validation. In this form, the number of folds is equal to N, the number of training examples. In each experiment, all but one of the samples is used as training data, and the test data consists only of the one remaining sample. The score of the algorithm is the percentage of experiments in which the test datum is classified correctly. This method is extreme in that the score for each of the individual experiments is based on only one sample, and therefore extremely insignificant, but many experiments are run so a significant total score is obtained. It is appropriate when the data set is extremely small, and as large a training set as possible is desired for each fold.

4 Causes of Overfitting

Before discussing ways of dealing with overfitting, let us look at some of the possible causes.

- A *small training set* is more likely to produce overfitting than a large training set. Patterns that show up in a small training set may be spurious, and due to noise. If they carry over to a larger training set, they are likely to reflect actual patterns in the domain.

- *Noise* in the data is likely to lead to overfitting. It increases the likelihood of spurious patterns that do not reflect actual patterns in the domain.

- Overfitting is more likely with a *rich hypothesis space*. Overfitting requires the ability to fit the noise in the data, which may not be possible with a restricted hypothesis space. For example, consider a hypothesis space consisting of conjunctive Boolean concepts (see last lecture), and suppose the true concept is actually a conjunctive concept but the training set is noisy. Since noise in the data is random, it is highly unlikely that a conjunctive concept will be able to classify the noise correctly, and so the conjunctive concept that performs best on the training set may well be the true concept.

- A domain with *many features* is more likely to lead to overfitting. Intuitively, each feature provides an opportunity for spurious patterns to show up. This is particularly an issue with *irrelevant features*, that are in the domain but have no impact on the classification. If there are many irrelevant features, it is quite likely that some of them will appear relevant in a particular data set. For this reason, some machine learning methods use *feature selection* to try to pick out the features that are actually relevant before beginning to learn.

5 Dealing with Overfitting

The basic idea behind virtually all methods for dealing with overfitting is to increase the inductive bias of the learning algorithm. Overfitting is the result of the algorithm being too heavily swayed by the training data. Increasing the bias means making stronger assumptions that are not supported by the data. This means in turn that more data will be needed to counter the assumptions. In particular, noise in the data, which will tend not to be supported by large amounts of data, will not override the inductive assumptions.

Increasing the inductive bias could mean using a more restricted hypothesis space than the original algorithm. In other words, the modified algorithm has a stronger restriction bias. For example, in using decision trees one could stipulate that only trees whose depth is at most 3, or that have at most 10 nodes, or use at most 4 features (or any combination of these) will be considered.

Alternatively, increasing the inductive bias could involve increasing the preference bias of the learning algorithm, so that some hypotheses will be preferred over others even if they perform worse on the training data. For decision trees, this means that simpler trees that have some training error will be preferred to more complex trees that have no training error.

This is the approach usually taken for decision trees. The general idea is to prune the tree learned by ID3, cutting off branches that are not justified by the data. Pruning means replacing a subtree with a leaf, whose label is the most common classification of the training instances that fall to the subtree.

There are two questions that still need to be answered: when to prune, and how to prune. For the first, there are two basic options. In *pre-pruning*, the algorithm stops growing the tree at some point, once it thinks growing the tree further would not be justified. Essentially, this approach adds another termination condition to the ID3 algorithm. In *post-pruning*, the algorithm grows the entire tree in the same way as usual, and only afterwards prunes away those parts of the tree that are not justified. The advantage of pre-pruning is that less work needs to be done, so it results in a quicker learning algorithm, while post-pruning has the advantage that more information is available for making the pruning decision. The decision is based on the entire subtree beneath a node, rather than just on the distribution of classes at a node.

We still have to answer the question of how to prune: when is the algorithm justified in growing a tree, and when should it stop growing? We will look at three methods. The first, *validation-set pruning* uses more data to determine whether a pattern in the training set is valid. The second, *chi-squared pruning* uses a statistical test on the data itself to determine whether an observed pattern is actually reflective of a pattern in the data, or the result of noise. The third, *minimum-description length* makes the tradeoff between the size of the tree and the amount of training error explicit. These three approaches, as used for decision trees, are paradigmatic of methods for dealing with overfitting in other learning frameworks.

6 Validation Set

The first pruning method tries to judge whether or not a particular split in a tree is justified, by seeing if it actually works for real data. The idea is to use a *validation set* — a portion of the training data set aside purely to test whether or not an actual split is actually a good split. In other words, the validation set is used to "validate" the model. A learned split is validated if it actually improves performance on the validation set. This provides a simple pruning criterion: prune a subtree if it does not improve performance on the validation set.

It is crucial when using this method that the validation set be separate from the data used to actually induce the decision tree. Otherwise the validation set itself will be used to create the tree, and so it will only be "validating" the pattern that was partly derived from itself in the first place. This will result in overfitting. Also, the validation set should not be part of the test data used to measure the performance of the learning algorithm. Clearly the validation set will indicate just the right amount of pruning when it itself is used as the test set, so using the validation set as the test set will inflate the accuracy score of the algorithm.

Using the validation set results in the following post-pruning algorithm, called *reduced error pruning*:

Build a decision tree T without pruning using ID3
For each node N of T, from the bottom up
  Let D_N be the training data that falls to N
  Let c be the most common class in D_N
  Let T' be the tree formed by replacing N with the leaf c
  If T' performs better on the validation set than T
    Replace N in T with leaf c // prune

7 Statistical Tests

The second approach to pruning is to apply a statistical test to the data to determine whether the pattern (i.e., distribution of classes) observed in the data that falls to a node is statistically significant. If it is not, the subtree beneath the node is pruned. This is a form of pre-pruning, since it is based only on the distribution of classes at the node, and can be made before growing the subtree itself.

A classical statistical test to measure whether the pattern exhibited by a distribution of classes is significant is the *chi-squared test*. Consider the null hypothesis that some attribute X is actually irrelevant to the classification. Suppose there are p positive and n negative examples in D, prior to splitting on X (we will consider Boolean classification for simplicity). Then if X is irrelevant, the correct model is that the class is positive with probability $\frac{p}{p+n}$.

We now ask the question: what is the probability that the actual distribution in the data would have been produced, if the null hypothesis that X is irrelevant was correct? The actual data can be characterized by p_x and n_x, the number of positive and negative examples that have value x, for each value x of X. Now, a perfect absence of pattern would correspond to the subsets D_x each having exactly the same distribution as in D. I.e., each subset D_x has $\hat{p_x} = \frac{p}{p+n}|D_x|$ positive examples, and $\hat{n_x} = \frac{n}{p+n}|D_x|$ negative examples. A measure of the total deviation of the data from complete absence of pattern is given by

$$\text{Dev}(X) = \sum_{\text{values } x \text{ of } X} \frac{(p_x - \hat{p_x})^2}{\hat{p_x}} + \frac{(n_x - \hat{n_x})^2}{\hat{n_x}}.$$

What does this mean? Dev(X) is just a number, called the *chi-squared statistic* of the data. The larger the chi-squared statistic, the higher the deviation from complete absence of pattern. So the method tells us to prune if the chi-squared statistic is too low. But what does "too low" mean? How do we scale the chi-squared statistic? The answer is to convert it into a probability.

Under the null hypothesis, splitting the data into v subsets will produce a chi-squared statistic that is distributed according to the *chi-squared distribution with v-1 degrees of freedom*. We ask, what is the probability, under the null

hypothesis, that splitting the data into v subsets will produce a chi-squared statistic $\geq \text{Dev}(X)$? If this probability is large, it means that the null hypothesis will often produce data with as much pattern as that observed. This implies that the existence of a pattern is not strongly supported by the data. If the probability is small, then the observed pattern in data is statistically significant. If the probability is p, we say that we reject the null hypothesis with confidence $1 - p$.

We can now make the chi-squared pruning method concrete. We need a *cutoff point* q, the required confidence needed to reject the null hypothesis and split on the attribute. We can now modify the ID3 algorithm by splicing in the following lines:

Choose the best splitting attribute X in $\mathbf{X}$
$\text{Dev}(X) = \sum_{\text{values } x \text{ of } X} \frac{(p_x - \hat{p_x})^2}{\hat{p_x}} + \frac{(n_x - \hat{n_x})^2}{\hat{n_x}}$
$v =$ number of values of X
$p = P(\text{chi-squared statisitic with } v - 1 \text{ degrees of freedom} \geq \text{Dev}(X))$
If $p > q$
  Then $\text{Label}(T) =$ the most common classification in $\mathbf{D}$
  Return T

One can use standard statistical software to compute the probability that the chi-squared statistic is at least a certain value. For example, the `chi2cdf` function in Matlab does the trick.

The advantage of the validation set approach is that it doesn't rely on unreliable statistical heuristics. The disadvantage is that it needs to reserve some of the training set (typically 10%-33%) for validation, instead of using it for learning the tree. For the most part, practitioners believe that the validation set approach works better in practice. In fact, Ross Quinlan, the inventor of ID3, began by using the chi-squared method, and later moved to the validation set method.

8 Minimum Description Length

We will briefly look at a third approach towards pruning. While it is not often used in practice for decision trees, it is a powerful method with an appealing theoretical basis that is used in a variety of learning algorithms.

Recall that pruning implements a strengthening of the preference bias of the learning algorithm, to prefer shorter trees even when they do not fit the training data as well. The idea of this approach is to make the tradeoff between the length of the tree and the fit to the training data explicit.

We define a "cost" function that measures the cost of a hypothesis as being based partly on the complexity of the hypothesis and partly on its training error:

$$\text{Cost}(H, \mathbf{D}) = f_1(\text{complexity}(H)) + f_2(\text{error}(H, \mathbf{D}))$$

for some appropriate functions f_1 and f_2. The Cost function makes the tradeoff

explicit by measuring the complexity and error components on the same scale. The question is how to choose f_1 and f_2 so that this makes sense.

Information theory provides an answer. A measure of the complexity of a hypothesis H is the number of bits needed to encode H. A measure of the error of the hypothesis H on the data $\mathbf{D}$ is the number of bits required to encode $\mathbf{D}$, given the optimal encoding scheme for hypothesis H. Both these measures are in the same dimension, number of bits.

A hypothesis H provides us with a way to describe data $\mathbf{D}$: first describe H, then use the optimal encoding scheme for H to describe $\mathbf{D}$. The total length of this description is the sum of the complexity and error terms. The *minimum description length (MDL)* principle says that we should choose the hypothesis H that minimizes this total.