CS 536 : Decision Trees

16:198:536

The Problem of Learning

As a general setup, we are given a collection of data points of the form $(\underline{x}, y)$ where $\underline{x}$ represents a vector of *features*, and based on those features we want to predict the value of y. The $\underline{x}$ could represent purchase history for various items, and y could represent whether or not a person purchased a separate item; the $\underline{x}$ could represent pixel values, and y could represent whether or not the image contains a dog. For the moment we limit ourselves to considering binary data and binary classification, that is $\underline{x} \in \{0,1\}^k$ and $y \in \{0,1\}$.

In general, we imagine having a collection of m-many data points $\{(\underline{x}^i, y^i)\}_{i=1,\ldots,m}$, drawn in an i.i.d. fashion from some underlying and unknown distribution. In the case of prediction or classification, given a *new* feature vector $\underline{x}$, we are interested in estimating the quantity

$$\mathbb{P}\left(Y = y | \underline{X} = \underline{x}\right), \tag{1}$$

i.e., the probability of belonging to a given class y given the feature data $\underline{x}$. This is generally a hard problem though it can be simplified somewhat using a Naive Bayes approach for estimating the unknown conditional probabilities.

To simplify things somewhat, we will assume that y is a deterministic function of the feature values, i.e., $y = f(\underline{x})$ for some unknown function f. Given the data points, the problem is then to estimate the function f. In general, we will assume that the function we are interested in constructing is a member of some **hypothesis space** H, and our goal is to construct some estimator $\hat{f} \in H$ given the data points.

The Problem of Overfitting

In general we would like to produce a function f that has low error. For classification problems, we can define error in terms of the probability of a mistake in classification:

$$\text{err}(f) = \mathbb{P}\left(f(\underline{X}) \neq Y\right), \tag{2}$$

i.e., the probability that f incorrectly determines the value of Y for a point drawn from the underlying distribution. The problem we are faced with generally is that if the underlying distribution of points is unknown, we cannot accurately assess this probability of error. Given the m-points of training data, we can only determine the average *training error*,

$$\text{err}_{\text{train}}(f) = \frac{1}{m}\sum_{i=1}^{m} \mathbb{1}\left\{f(\underline{x}^i) \neq y^i\right\}. \tag{3}$$

This serves to count the average number of mistakes f makes over the training data.

In an ideal world, if we construct an f with a small value of $\text{err}_{\text{train}}$, it would have a small value of err. This is not necessarily the case, however. Suppose that f^* represents the 'true' function, and take H to be the set of all binary functions on k variables. In this case, $f^* \in H$ produces zero training error and zero true or testing error. However, H is large enough that there will be functions $f \in H$ that agree with f^* on each of the m training examples, and no place else! As a trivial example,

$$f(\underline{x}) = \begin{cases} f^*(\underline{x}) & \text{if } \underline{x} = \underline{x}^i \text{ for some i,} \\ 1 - f^*(\underline{x}) & \text{else.} \end{cases} \tag{4}$$

For the above, f would produce zero training error, but would produce the incorrect answer on every data point $\underline{x}$ that was not in the original training set. If all we have to go on when constructing the function is the training set, then there would be no way to determine which was a better fit to the data, f or f^*.

In general, we want whatever f we construct to **generalize** well to data that we have yet to see. That is, we want to avoid the situation where $\text{err}_{\text{train}}(f)$ is small, but $\text{err}(f)$ is large - in this case we say that f is *overfit* to the data and fails to generalize. An easy way to do this would be if we have training data that covered every possible instance we might be faced with - in which case we would have that $\text{err} = \text{err}_{\text{train}}$. This will generally be infeasible though, requiring an unreasonable number training data points. We can quantify this somewhat more precisely with the following result:

For a finite hypothesis space H, if we want $|\text{err}(f) - \text{err}_{\text{train}}(f)| < \epsilon$, we need at least m training data points where

$$m \geq \frac{1}{\epsilon^2} \log(|H|). \tag{5}$$

Note: We will quantify this result more precisely when we get to statistical learning theory, but generally speaking it says that in order to avoid overfitting and ensure that f generalizes, the more hypotheses we are comparing the more data we need in order to differentiate them. And the greater the accuracy we want, the more samples we need to achieve that accuracy. The intuition here - in the case of binary data - is that every data point should ideally be able to eliminate half of the possible hypotheses, and thus to determine the true best hypothesis we need enough data to eliminate all but one, i.e., $O(\log(|H|))$ data points.

Applying this in the case of H as the set of binary functions on k variables, we have that $|H| = 2^{2^k}$, in which case the result states we would need $m \geq O(2^k)$ data points for training - effectively stating we would need to see almost every possible value of $\underline{x}$ in our training data.

This suggests that on technical reasons alone, we cannot define the problem of learning simply to be solving

$$\hat{f}^* = \text{argmin}_{f \in H} \text{err}_{\text{train}}(f). \tag{6}$$

This is typically solved by *regularization* - effectively constraining the set of hypotheses that we are willing to consider. Rather than, in this case, consider the full set of binary functions on k variables, we might limit ourselves to functions of a specific form. We can justify this in one of a couple of ways:

- **Occam's Razor:** *Non sunt multiplicanda entia sine necessitate* - Entities are not to be multiplied without necessity. Or, equivalently, you should not make explanations any more complicated than they need to be. All things being equal, the simpler explanation is usually the right one.
- Additionally, the more complex a hypothesis is (the more entities it takes into consideration), the more likely that it is describing noise or spurious relationships that don't actually tell you anything. Again this is a general rule of thumb, but it is certainly true that the more complicated a hypothesis is, the more *opportunities it has* to describe noise or relationships that don't actually exist.
- The fewer the hypotheses under consideration, i.e., the smaller the value of $|H|$, the less data we need in order to distinguish between them.

Regularization renders the problem of learning feasible.

In the remainder of these notes, we consider the problem of learning *decision trees*, a specific functional form for the hypotheses in H. This will require defining decision trees, determining how decision trees can be learned or fit to data, and specifying a method of constraint or regularization, to try to ensure that the resulting models generalize well.

Decision Trees

A **decision tree** is a particular form function effectively capturing the idea of nested if statements:

```
f(x1, x2, ..., xk) =
    if xi = 0
        if xj = 1
            ....
        else
            ....
    else
        if xl = 0
            ....
        else
            ....
```

where each '...' indicates potentially more nested if statements, each eventually terminating in some return function, specifying a value based on the results of all the comparisons up to that point. This is frequently visualized in terms of a **tree**, where the root of the tree indicates an initial decision based on some variable x_i, and if $x_i = 0$ the function proceeds down one branch, otherwise if $x_i = 1$ it proceeds down the other, each branch leading to further nodes and decisions to make based on the value of indicated variables. The terminal nodes or leaves of the tree represent final return values for y.

Decision trees (on binary data) are a specific way of expressing a binary function, but they are in fact as expressible as binary functions are generally. To see this, note that the path from the initial root node to the final terminal leaf node effectively expressed a conjunction of the variables traversed - in this way, the Disjunctive Normal Form (DNF) of any boolean function can be expressed as a decision tree, organizing the variables along the tree traversal to terminate as appropriately in the necessary conjunction. In this way, decision trees are potentially as expressive as boolean functions generally - but they admit a relatively straightforward learning algorithm based on this recursive structure. In particular, to fit a decision tree to a collection of data points:

- Select some variable X_i
- Partition the data into two sets, all the data points where $X_i = 0$ and all the data points where $X_i = 1$
- Recursively construct a decision tree on the $X_i = 0$ data, and on the $X_i = 1$ data
- If, at any point, the remaining data down one branch has the same Y value, set that node to return that Y value
- Recurse until all data is capture by the terminal node of some branch on the resulting decision tree

This gives a relatively straightforward process for generating decision trees according to data, and assuming there is no contradictory data in the data set (for instance $\underline{x}^i = \underline{x}^j$ but $y^i \neq y^j$) will be able to fit a decision tree to the data with zero training error. However this process is not unambiguous - there is flexibility for instance in how the splitting variable X_i is selected at every step. In order to address this, we need to specify what would make one choice of splitting variable better than another.

In particular, this is the point where **regularization** can be applied to the problem of learning decision trees. As stated, decision trees are as expressive as boolean functions generally, and so the problem of over fitting is very real

here. As stated, the above process can result in a tree with zero training error, but we want to try to ensure good generalization as well. One potential way to regularize in this case, or to quantify the notion of 'complexity' for decision trees is through decision tree depth. In particular, the deeper any one branch is, the more variables that factor into the final terminal node. Shorter branches, over all, will correspond to terminal nodes that factor in fewer variables. From this perspective, 'simple' trees that fit the data will generally be shorter, and we could consider the problem of trying to find the shortest possible tree to fit the data.

However, in full generality, identifying the shortest tree that fits a data set is a computationally difficult problem. Therefore it is difficult to know with certainty at every step what variable is with splitting on in growing your decision tree. As is frequently the case, in the face of computationally hard problems, we find great comfort and utility in falling back to heuristics and approximations. While these may not provide the 'best' result in terms of optimizing with respect to depth, they are frequently quite good.

Building Decision Trees

The **ID3** Algorithm for building decision trees follows the previous outline, but at every step chooses the variable to split on based on the following heuristic: of the available variables X, which one provides the most information about the value of Y?

We can quantify this using the notion of **information content**. In general, the information content of a (discrete) random variable Y is defined to be

$$H(Y) = -\sum_y \mathbb{P}(Y = y) \log \mathbb{P}(Y = y), \tag{7}$$

where the sum is taken to be over the possible outcomes of Y. This can be thought of as a way to quantify the uncertainty in the outcome of Y. Imagine for instance that Y were equally likely to be any number between 1 and 1024 - in this case, $H(Y) = -\sum_y (1/1024) * \log(1/1024) = 10$, corresponding to maximum uncertainty between the 2^{10} possible outcomes. Imagine, however, that $Y = 1$ with probability 1, and $Y = 2, \ldots, 1024$ with probability 0. In this case, you can show that $H(Y) = 0$, corresponding with maximum certainty - you are confident, with probability 1, that you know the true value of Y. This is frequently interpreted as saying that (in this case), you need on average 10 bits to express the value of Y in the case of maximum uncertainty, but in the case of maximum certainty you actually need zero bits, because the value of Y is effectively known. In general, $H(Y)$ can be interpreted as the average or expected number of bits to encode the outcome of Y, observing that the more confident you are in an outcome occuring, the fewer bits you typically need to express it.

If $H(Y)$ indicates the information content of Y, we can quantify how much 'X tells us about Y' in the following way: the conditional information content of Y given $X = x$ is given by

$$H(Y|X = x) = -\sum_y \mathbb{P}(Y = y|X = x) \log \mathbb{P}(Y = y|X = x). \tag{8}$$

In short, conditioned on the fact that $X = x$, how much certainty or uncertainty is there in the value of Y?

We can generalize this further in the following way - on *average*, regardless of actual outcome, what does X say about Y? In this case, we can define the conditional information content of Y given X as the expected value or average of the previous:

$$H(Y|X) = \sum_x \mathbb{P}(X = x) H(Y|X = x) = \sum_x \mathbb{P}(X = x) \left[-\sum_y \mathbb{P}(Y = y|X = x) \log \mathbb{P}(Y = y|X = x) \right]. \tag{9}$$

With this tool, we can now define the 'information gain' of X - how much do we know now about Y that we didn't know previously:

$$\text{IG}(X) = H(Y) - H(Y|X). \tag{10}$$

Note that if X completely determines Y, that is if X is known we have total information and total certainty about Y, then $H(Y|X) = 0$, hence $\text{IG}(X) = H(Y)$: X tells us everything there is to know about Y. In the case that X is *independent* of Y, you can show that (since $\mathbb{P}(Y|X) = \mathbb{P}(Y)$), that $H(Y|X) = H(Y)$ and hence $\text{IG}(X) = 0$: X tells us nothing we didn't already know about Y. Using this observation, **ID3** proposes the following greedy method for building decision trees:

The ID3 Algorithm:

- For each variable X_i, compute $\text{IG}(X_i)$
- Select the variable with maximum information gain
- Split/Partition the data based on that variable
- Recursively apply this algorithm on each part of the data

Note! It is worth noting that while the information gain of a variable is well-defined, we can really only compute it if we know the true underlying probabilities for each variable. This is of course impossible, since the distribution of the data is unknown, so ID3 accounts for this by *approximating* the probabilities with the relative frequencies when computing the information gain.

In general, ID3 proceeds by trying to greedily order the variables in terms of information gain in the hopes that his leads to short trees over all. It is not the only such algorithm (see CART and the Gini Impurity), but in general it is quite effective and can be implemented in an efficient way.

Evaluating and Pruning Decision Trees

ID3 (and related ideas) give a means for constructing decision trees. In general, you can easily imagine taking a data set, applying ID3, and running it to completion to generate a decision tree. However, given a decision tree, there are two interesting problems that remain:

- How can we evaluate how good a decision tree is?
- Can a decision tree be improved after the fact?

As noted previously, given a learning problem, we can really only evaluate the training error - how well the algorithm performs on the data we have available. The true underlying distribution of the data is hidden from us, and therefore it can be difficult or impossible to determine $\text{err}(f)$.

One potentially approach to this problem is to devote only a fraction of the data (say 80%) to training, and set aside the remaining data for testing and validation. In essence, by setting aside a portion of the data that the model was not trained on and has never seen, we can use this portion of data to approximate the underlying distribution and estimate the true error. The error of a model on its testing data could be reported for instance as an estimate for the true performance error. Additionally, if it is found that a model has good training error, but poor testing error on the testing data - this can be taken as a sign that the model has become over fit, and potentially needs to be tossed.

An important proviso: This last step, using the testing data to evaluate whether or not a model has become overfit, is potentially risky in that it allows your learning algorithm to implicitly 'peek' at the testing data. In general, testing data needs to be isolated from the training process and should not be used to inform the model. In order to work around this, frequently a third portion of data or *validation* data is set aside, with the understanding that it is not to be looked at until the final step of evaluating your final model, and estimating its true error. If the validation data is additionally used as a check for over fitting, we completely lose the ability to assess our model beyond the training error.

Generally speaking, while ID3 (and related algorithms) give a bias towards shorter models rather than longer models, it is still possible that the algorithm can generate unnecessary complexity and spurious correlations. It is useful then to consider **pruning** decision trees - taking a decision tree and removing potentially unnecessary or unimportant decision steps to try to simplify and improve generalizability. There are four ways that you might consider doing this:

- **By Depth:** Again, the depth of the decision tree generally reflects the overall complexity of the model. This can be utilized in building decision trees by specifying a maximum depth that you are willing to consider. Once you reach that depth in the construction, even if the remaining data set down that branch is not 'pure', you terminate, deciding Y for that terminal node perhaps by the majority decision of the remaining data down that branch, or assessing probabilities based on the relative fraction of Y values. It can be difficult to know in advance though how deep a reasonable threshold is, however.

- **By Sample Size:** In general, as the number of data points goes down (as it will for each branch, during the recursive split and construct procedure), you can be less confident in any decisions based on that data. One common practice is to set a threshold value and once the number of remaining data points falls below that point, terminate growth for that branch and determine a value of Y for that terminal node again by the majority of remaining data. It can be difficult to know how large sample sizes need to be in order to be confident of results, however. This is expanded slightly in the following idea.

- **By Significance:** As noted, if X is independent of Y, then X provides no useful information about Y and cannot be usefully utilized in a split. However, we are in a position where we don't know the true underlying distribution and hence cannot verify completely this hypothesis of independence. In particular, natural fluctuations in data can potentially mask independence and give positive information gains. One way around this is utilizing the idea of the χ^2-test for independence. This is a statistical test that allows you to assess, given a set of occurrence data, how likely this data would be if the variables you are studying were actually independent. See related notes for a more in depth analysis of this. This can be utilized as a pruning mechanism by testing a chosen split variable to determine if there is data to support the idea that it is independent. If the threshold for significance is not met, that variable can be discarded as a split variable . This helps limit the extent to which the decision tree learns noise and spurious relations in the data.

- **Early Termination:** This is a technique for stopping growth of the tree early, before noise or spurious information is added to the tree. In general, we expect that as the tree grows, the error on the training set should go down. Ideally, it will go all the way down to zero, but in general (for instance if there is contradictory data), it should still decrease with added complexity. However, the error on a portion of testing data that has been set aside should behave differently - ideally, starting out it should decrease (as the tree learns important relationships in the training data), but as the tree starts to overfit the training data and lose generalizability, you frequently observe error on the testing data start to increase. Early termination simply says to watch how these two errors change as the depth or complexity of the tree increases, and stop tree growth the moment the testing error starts to increase (or a certain threshold is met). Note, implementing this will require additional

testing or validation data to be set aside since the training algorithm is explicitly peeking at the testing data, but this can be a very useful heuristic for informing decision tree growth.