Decision Trees

CS 360 Machine Learning
Week 3, Day 2

February 8, 2024

Contents

1 Finishing up cross validation

A few other things to discuss from cross-validation. One thing you can do is use cross-validation to choose hyperparameters. First, let's think about the difference between parameters and hyperparameters. Thinking about polynomial regression, the hyperparameters are the degree of the polynomial, while the parameters are the weights on each features. For logistic regression, the hyperparameters would be the learning rate, max iterations, etc., while the parameters are again the weights since they're automatically learned once we choose the hyperparameters. For k-nearest neighbors the hyperparameters are k and the distance metric.

Hyperparameters are hard to define precisely, but are a parameter that controls other parameters. We can't choose them based on the test data since that breaks our cardinal rule: **don't look at the test data!** But we can choose them based on validation data.

One common way to do that is to use *grid search* with cross-validation. In this case, we set the hyperparameter choices we're testing, train all splits, and look at the average for that hyperparameter set over all splits. We then choose the set of hyperparameters with the highest average over all splits. This allows us to choose the hyperparameters in a more principled way.

In *leave-one-out* cross validation, you use $n - 1$ examples to train and then test on the remaining example. This is mostly used for small datasets. You can also use multiple trials of leave-one-out cross validation where you use different splits each time. Whether using multiple trials or not, you would similarly look at the hyperparameter set that achieves the highest average accuracy over all splits.

In reality, people will often only do one split of their dataset into training and validation data. This may partially be motivated by concerns about honesty—how do you know that someone didn't look at their test data? So sometimes the validation data performance is actually what's reported, but this isn't ideal.

2 Decision Trees

You might recall from CS 260 that we saw decision stumps. Decision trees extend that to a higher depth where many features are used in combination. These are used in the real world in medical diagnostics or other scenarios where it's important that the model be directly examinable. Another way to use them is to model another model, for example use decision trees to interpret the results of a neural network.

Consider one example—deciding whether to play tennis on a specific day based on weather, humidity, and wind. The labels at the leaves of the tree say whether you play tennis on that day or not. In general in a decision tree, the leaves give the resulting prediction y (e.g., the class resulting from the classification). We think about internal nodes as testing one feature at a time, with the branches coming from that node representing the possible values for that feature.

The root of the tree is at $depth = 0$. Moving down, the depth increases by one when following each branch of the tree. Note that these do not need to be binary trees; the branching factors will depend on the different values the features can take.

Suppose that someone gives us a decision tree and we want to classify a test example. We would start at the root node and check the feature value of the feature available at the root, following the branch with the value of the feature in our test example all the way down to the leaf node predictions. Decision tree predictions can be made fairly quickly — it's based on the time it takes to traverse from the root to the leaf nodes, i.e., the depth of the tree.

What about in the case where we have continuous features? Our goal here will be to partition the data space. This has some similarity to splitting the space for *kd*-trees, but we'll choose our partitions differently based on the point labels. The node splits will then be based the feature values (e.g., $x \leq 3$) with resulting regions having consistent predictions.

Decision trees are good because they're very interpretable; it's easy to identify *why* a classification was made, for example by looking at the choices on the path from the root node to the leaf node for a specific example. Decision trees can also generate compact representations and fast predictions. However, decision trees can also be very brittle to specific feature choices since they don't look at each example holistically, and getting the implementation done well to avoid overfitting can be difficult. Continuous features must also be discretized, and how to do this is not always clear. Too many splits can also lead to overfitting.

Decision trees are flexible enough to handle both non-binary feature values and non-binary class labels. They can also be used for regression, for example averaging the training examples found at a leaf to predict the resulting value.

For high depth trees, there's a concern that the tree may be overfitting, i.e., it may have essentially memorized the training examples that appear at the leaves. We'll talk more in a bit about how to determine whether a tree is overfitting and how to avoid it.

2.1 ID3 decision tree algorithm

The basic idea to build the tree using this algorithm is that we want to partition the data based on the feature value that best informs the prediction of the resulting $\hat{y}$. So first we'll select the best feature, divide into partitions based on the value of this feature, and then recurse on those subtrees.

For any subtree, we consider the below pseudocode for that subtree, where D is the dataset (X, y) and F is the feature names we'll use for the tree.

```
MakeSubtree(D, F):
    if stopping criteria met
        make a leaf node N
        determine class label/ probabilities for N
    else
        make an internal node N
        S = FindBestFeature(D, F)
        for each outcome k of S:
            Dk = subset of instances that have outcome k
            N.child[k] = MakeSubtree(Dk, F-S)
    return subtree rooted at N
```

A few points to consider from the above pseudocode. First, when considering how to determine a class label for the leaf node (4th line above), we'll use the majority class over the examples at that node for this lab.

When you consider this line:

```
            N.child[k] = MakeSubtree(Dk, F-S)
```

why don't we use the feature again (why $F - S$)? Since you just split on that feature, you've already limited the values based on that feature, so it won't be informative, but you might accidentally choose it again which could lead to an infinite recursive call.

How do we determine the stopping criteria? Suppose that all the data points in the partition have the same label. There's no point in going further—we should just predict that label! Or we could decide that, e.g., 95% of the points having the same label is close enough and stop then. (Modifications like this could help us to avoid overfitting.) We might also have a case where there's no more features remaining to split on, i.e., we've used each once before removing it from the set. Or we might have reached a case where no more features are informative for the label, i.e., all the feature values in our partition are the same for all examples— in this case there's no point in splitting further. Or finally, you might have reached a user-specified (hyperparameter) maximum depth in the tree—we'll use this option in the lab.

2.2 Implementation suggestions

Suppose that we have a dataset with all the training examples. In order to create a partition, it's useful to have an associated partition data structure that has a subset of the data with the relevant feature values. For example, if a node based on weather has features sunny, overcast, and rain, we'd want three partitions and in each of those subsets the feature for weather would have only one value (either sunny, overcast, or rain). Additionally, since those feature values would be constant for that partition, we could essentially ignore that feature value entirely—we could even remove that feature from the possibilities. It's useful to store the features as a dictionary where the key is the feature name and the value is the feature's value.

Deciding how to choose the best feature should be done based on the entropy (denoted $H(X)$) for that feature. When choosing the root (or a subtree root) node, we want to choose a feature that will give us the most homogeneity in the labels for each resulting subtree. We want there to be a correlation between the feature and the label we're picking. The basic steps for this are to 1) think about the overall probability of the label in the data and the associated label entropy $H(Y)$ in the data and 2) look at the conditional entropy $H(Y|X)$, i.e., the extent to which the information about the feature contributes to lowering the entropy of the label. We want to choose the feature that minimizes the conditional entropy

$H(label|feature)$; this is equivalent to thinking about the maximum information gain $H(Y) - H(Y|X)$ about the label given the feature. We won't be implementing the entropy calculation for the lab, but will be using it to choose features.

A few additional implementation hints:

- Recall that these trees are not always binary.
- Your prediction or classification algorithm is also recursive; i.e., the prediction / query algorithm on a specific example is recursive.
- You can parse the feature name to figure out if it's continuous or discrete and how to classify, e.g., "age <= 45."