Introduction to Machine Learning **Fall Semester, 2013**

Lecture 12: January 5

Lecturer: Yishay Mansour *Scribe: ym*

12.1 Decision Tree: Pruning

12.1.1 Why Pruning?

In the previous lecture we saw the algorithm that builds a decision tree based on a sample. The decision tree is build until zero training error. As we saw before, our goal is to minimize the testing error and not the training error.

In order to minimize the testing error, we have two basic options. The first option is to decide to do *early stopping*. Namely, stop building the decision tree at some point. Different alternatives to perform the early stopping are: (1) when the number of examples in a node is small, (2) The reduction in the slitting criteria is small, (3) a bound on the number of nodes, etc.

The alternative approach is to first build a large decision tree, until there is no training error, and then prune the decision. The net effect is the same. In both cases we end with a small decision tree. The major difference is what we observe on the way. When we first build a large tree, we may discover some important sub-structure that in case we stop early, we might miss. In some sense, we can always simulate the early stopping when we first build the large tree and then prune, but definitely cannot do the reverse.

12.1.2 Decision Tree Pruning: Problem statement

The input to the decision tree pruning is a decision tree T and a sample $S = (S_1, S_2)$, where we use S_1 to build the decision tree T.

The output of the pruning is a decision tree T' which is derived from T. We will mainly look at the case where we can replace an inner node of T by a leaf. Another, more advanced option, is to replace an inner node by the sub-tree rooted at one of its children. At the extreme, if we have no restriction then we are essentially left with the same problem we started with, finding a good decision tree.

12.1.3 Reduced Error Pruning

In reduced error pruning we split the sample S to two part S_1 and S_2. We use S_1 to build a decision tree and use S_2 is used for decisions of how to prune it. We do not assume anything

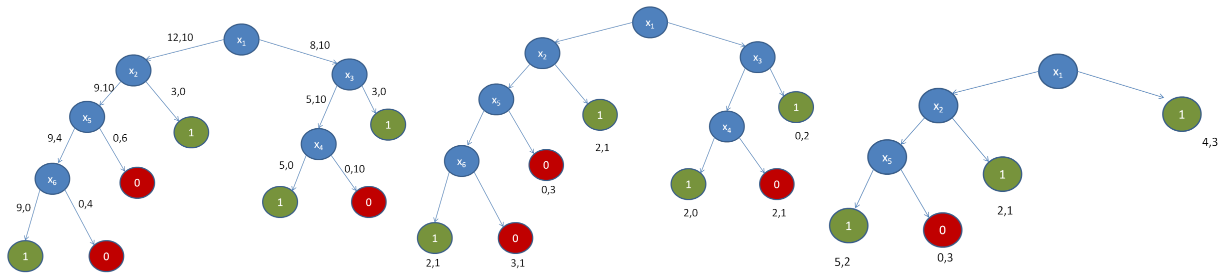

Figure 12.1: Example of data for decision tree Pruning. The left figure is the building of the decision tree using S_1. The middle is the set S_2 used for T. The right is the pruned decision tree. The numbers are the number of examples labeled by 1 followed by the number of examples labeled by 0.

about the building of the decision tree T using S_1. (We do not even assume that the decision tree T has zero error on S_1, although in our main application this will be the case.) Our pruning will use S_2 to modify the structure of T. The pruning of an internal node v is done by replacing the sub-tree rooted at v by a leaf.

In the pruning we traverse the tree bottom-up and for each internal node v we check whether we want to replace the sub-tree originated by it with a leaf or not. The decision of whether to prune the node v or not is rather simple: we sum the errors of the leaves in the sub-tree rooted by node v and the errors in case we replace it with a leaf. If replacing the sub-tree rooted at v will reduce the total error on S_2 we prune v. In case the number of errors is the same we also prune the node in order to minimize the size of the decision tree we output. Otherwise, we keep the node (and the sub-tree rooted at it).

Notice that on each step in the pruning algorithm we examine the sub-tree that is relevant to the current stage in the algorithm, meaning that if we pruned part of the sub-tree rooted in v on previous iterations then when we check v we examine it against its current sub-tree and not the original one. The pruning of an internal node v is done by replacing the sub-tree rooted at v by a leaf at v.

12.1.4 Model Selection

In this section we present an algorithm that produces few pruning possibilities. The algorithm will be wrapped by a model selection procedure (either Structural Risk Minimization, Cross Validation, or any other) that chooses one of the pruning possibilities produced by the underlaying algorithm. Generally the core algorithm will be given a decision tree T and target number of errors e and will produce the smallest pruned trees that has such errors rate.

Algorithm

1. Build a tree T using S.
2. for each e, compute the minimal pruning size k_e (and a pruned decision tree T_e) with at most e errors.
3. Select one pruning using some criteria.

We will first show how to find the pruning that will minimize the decision tree size for a given number of errors. (Or alternatively, find the pruning that for a given tree size minimizes the number of errors.)

Finding the minimum pruning

Given e, the number of errors, we want to find the smallest pruned version of T that has at most e errors. We give a dynamic programming algorithm. We use $\mathbf{T}[0]$ and $\mathbf{T}[1]$ to represent the left and right child subtrees of $\mathbf{T}$ respectively. We denote by root($\mathbf{T}$) the root node of the tree $\mathbf{T}$. We also define $\mathbf{tree}(r, \mathbf{T_0}, \mathbf{T_1})$ to be the tree formed by making the subtrees $\mathbf{T_0}$ and $\mathbf{T_1}$ the left and right child of the root node r. For every node v of $\mathbf{T}$, $\mathbf{Errors}(v)$ is the number of classification errors on the sample set of v as a leaf.

```
prune($k$:numErrors, $\mathbf{T}$:tree) $\Rightarrow$ ($s$:treeSize, $\mathbf{P}$:prunedTree)
    If $|\mathbf{T}| = \mathbf{1}$ Then                    /* Test if $T$ is only a leaf */
      If $\mathbf{Errors}(\mathbf{T}) \leq k$ then
        $s = 1$
      Else
        $s = \infty$
      $\mathbf{P} = \mathbf{T}$
      Return
    If $\mathbf{Errors}(\text{root}(\mathbf{T})) \leq k$ Then                /* Keep root as leaf */
      $s = 1$
      $\mathbf{P} = \text{root}(\mathbf{T})$
      Return
    For $i = 0 \ldots k$                    /* Check for $i$ and $k - i$ errors at children */
      ($s_i^0$,$\mathbf{P}_i^0$) $\leftarrow$ prune($i$,$\mathbf{T}[0]$)
      ($s_i^1$,$\mathbf{P}_i^1$) $\leftarrow$ prune($k - i$,$\mathbf{T}[1]$,)
    $I = \arg\min_i \{s_i^0 + s_i^1 + 1\}$
    $s = s_I^0 + s_I^1 + 1$
    $\mathbf{P} = \mathbf{MakeTree}(\text{root}(\mathbf{T}),\mathbf{P}_I^0,\mathbf{P}_I^1)$
    Return
```

Note that given the values for the two child subtrees, we can compute the values of the node. The basic idea is therefore to do the computation from the leaves up towards the root of the tree. The running time an be computed as follows. At each node we need to go over all values of $i \in [0, k]$, namely $k + 1$ different values. For each value of i we need to compute the sum of the size of the pruned left subtree with i errors and the size of the pruned right subtree with $k - i$ errors. This is done in $O(1)$. Building the pruned decision tree is also $O(1)$ since it involves only pointer manipulation. (If we will duplicate the pruned tree then it would be at most $O(|T|) = O(m)$.)

Assume that we have a sample size of m. This implies that the decision tree is of size at most $O(m)$. The overall running time is $O(m^2)$, since $k \leq m$ and the running time of each invocation is $O(k) = O(m)$.

Model Selection using Cross Validation

We use a held-out set S_2. We construct the different pruning P_i, where $i \in [1, m]$. We will select between the P_i using S_2. Namely, we will select the P_i which has the least number of errors on S_2.
How large should S_2 be?
Similar to selecting a hypothesis from a finite class of m hypothesis (in the PAC model), if we set $|S_2| = \frac{1}{\epsilon^2} \log \frac{m}{\delta}$ then with probability $1 - \delta$ we have for each P_i a deviation of at most ϵ in estimating the error of P_i. This implies that with probability $1 - \delta$ the pruning P_i which minimizes the observed error of S_2 has true error of at most 2ϵ more than the true error of the best pruning P_j.

Model Selection using Structural Risk Minimization

We have a set of prunings P_i, for $i \in [1, m]$. We will select between them using the existing examples, and not use another set of examples. The Structural Risk Minimization (SRM) will use the following decision rule,

$$T^* = arg \min_i \{error(P_i) + \sqrt{\frac{|P_i|}{m}}.\}$$

Model selection - summary

The main drawback of this approach is the running time, which is quadratic in the sample size. This implies that for a large data set the approach will not be feasible. The benefit is that in case of small data sets it allows to utilize much better the examples we have.

12.2 Ensemble Methods

12.2.1 Basic idea and rational

The idea behind ensemble methods is very simple, combine multiple hypotheses to build a (hopefully) more accurate hypothesis. The two most important questions that we should answer, when considering a specific ensemble method are the following:

1. How do we generate the multiple hypotheses. Recall that we have only one sample, so we need to specify how to use a single sample to generate multiple hypotheses.

2. How do we combine the hypotheses. Once we have multiple hypotheses we need to specify how do we combine them to a single prediction. The most natural is a majority rule or a weighted majority rule, but many other alternatives exists.

There are a few a basic reasons why ensemble methods can offer an advantage.(See also Figure 12.2.)

1. **Statistical reasoning:** The sample does not have enough information to tightly specify the target hypothesis. There is a significant uncertainty about which hypothesis is the "right" one. One can envision that we have multiple good hypotheses to better specify the target function. This reason stresses the fact that we have limited amount of data.

2. **Computational reasons:** Many times we have to resort to heuristic methods when searching for the best hypotheses given the data, given computational constraints. The EM algorithm is an excellent example of this computational limitation. This suggests that generating multiple hypotheses might give a better approximation of the target function. This reason stresses the fact that many times we have a limited amount of computation power.

3. **Representation reasons:** If the space of hypotheses is not convex, then averaging multiple hypotheses might represent a hypothesis which is outside our hypotheses class. This is true even with infinite data and unlimited computational power.

12.2.2 Boosting

Boosting is actually an ensemble method. In boosting we are generating different hypotheses by changing the sample distribution, and reweighing the examples. Based on the weak-learner hypotheses we are guarantee to generate different hypotheses.

In boosting we combine different the hypotheses using weighted linear threshold. The coefficient of the different hypotheses are determine when the weak learners are selected.

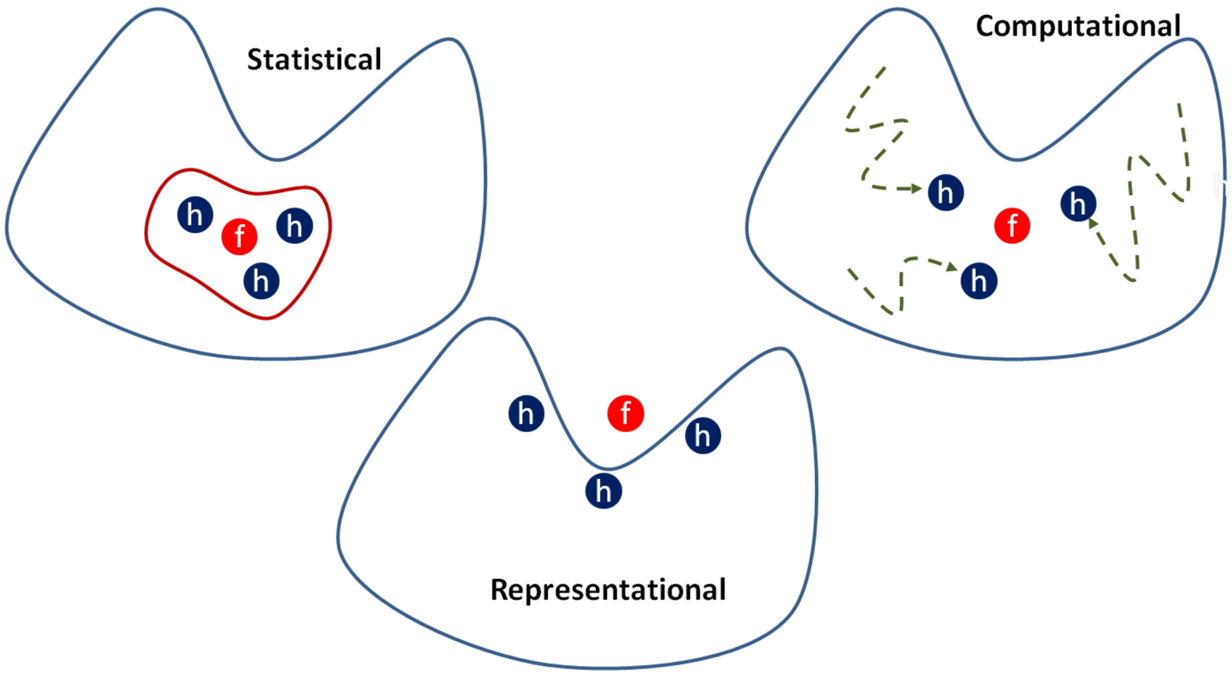

Figure 12.2: Reasons for using ensemble methods

12.2.3 Bagging

The idea behind bagging is to generate different samples and use them to learn different hypotheses. The problem is that many times we do not have a huge data set and we like to make the best use of it. For this reason we will sample the data set multiple times to generate the different hypotheses.

More precisely, in *bagging* the inputs are a single learning algorithm A and a sample S. We perform the following steps:

1. Given S we generate sub-samples $S_1, \ldots, S_k$ by selecting items from S with repetitions. (The repetitions guarantee that each example by itself has the same distribution as the underlying distribution.)

2. Given $S_1, \ldots, S_k$, we run A on each S_i and learn a hypothesis h_i.

3. We combine the different hypotheses using a simple majority.

A good question is *why are we making progress*. Note that the expected error of each generated h_i is identical (before sampling S_i), and slightly higher than learning on S (since we use less examples). This error can be viewed as a *bias* which is inherent between the target function and the learned hypotheses, given the sample.

The main gain in averaging is done by reducing the *variance*. The variance of a error single hypothesis fluctuates considerably. In contrast, the majority of many hypotheses is much more stable. This suggests that we should be getting better generalization bounds.

12.2.4 Stacking

We like a general methodology of combining multiple hypotheses. For example we might learn a decision tree, a large margin classifier and AdaBoost. How can we combine them? Majority is only one option!

Stacking give a general methodology of combining multiple hypotheses. The input is as sample S, combining algorithm C, and k learning algorithms $A_1, \ldots, A_k$. We run the staking procedure as follows.

1. We run algorithm A_i on S to generate hypotheses h_i.
2. Given $h_1, \ldots, h_k$, we build a new sample S', such that for any $(x, y) \in S$ we have $((h_1(x), \ldots, h_k(x)), y) \in S'$.
3. We run C on S' to generate the hypotheses H.

We can view bagging as a special case of stacking where A_i sub-samples S and C is a simple majority. Similarly, boosting can be also viewed as a special case of stacking, where A_i generates the ith weak hypotheses, and H is the weighted majority using the coefficients α_i. Random forest, presented in the next section, is another example of stacking.

12.2.5 Random Forest

Decision trees construction is highly sensitive to the sample. A tiny change in the sample can generate a different predicate in the root, which will result in a completely different decision tree. In bagging we tried to overcome this problem by generating multiple decision tree and taking their majority. In the original paper a sub-sample of about 66% was observe to be the best in those experiments. This is one source of generating different decision tree.

Another source of generating different decision tree, is selecting which subset of attributes will be considered in a given node. There is a parameter M which controls how many of the N attributes we will consider. When $M = N$ we consider all attributes, and we are back to the regular decision tree algorithm. For $M < N$ we select a random subset of M attributes, and consider only them as candidates for the current node. We select the attribute that minimizes the splitting index function. Note that in different nodes we select different subsets randomly! Also, note that for $M = 1$ we simply generate a random tree.

To combine the multiple decision trees, similar to bagging, we are using a simple majority rule.

The main benefits of the random forest are: (1) Fast to run, (2) Easy to parallelize, and (3) Competitive performance with leading machine learning algorithms (AdaBosot and SVM).

The main weaknesses are: (1) Random forest losses the interpretability of decision trees, (2) Many parameters around that need to be tuned, and (3) Feature selection collides with sampling attributes.