Selected Topics on Decision Tree Models and Related Ensembles

CS109a/CS209a/Stat 121 Advanced Section
Instructors: Pavlos Protopapas, Kevin Rader, Rahul Dave, Margo Levine
TF: Patrick Ohiomoba

Introduction

In class we've learned about decision trees in general with a specific focus on the general properties of classification and regression trees. At this moment, ensemble approaches that take a "committee" approach of combining many individual predictors in the hopes of more effective predictive capability have overtaken decision trees in popularity. The aim of this section is to gain deeper insight into a selection of properties of decision trees that may help in understanding and manipulating tree-based ensemble methods — in particular random forests and gradient boosting — some essential characteristics of which follow from the decision tree frameworks upon which they're built.

Notation

We'll for the following discussion assume unless otherwise specified a supervised learning problem in which the data comes as a finite learning set $\mathcal{L} = (X, y)$ with X being the input samples of predictors and y the output values, and that our ultimate goal is to build an estimator $\phi_{\mathcal{L}} : X \to y$ such that

$$Error(\phi_{\mathcal{L}}) = E_{X,Y} L(Y, \phi_{\mathcal{L}}.predict(X))$$

is minimized for the loss function L.

In describing decision trees we'll use t very often to describe nodes and T to denote trees. We'll use s to denote splits and i(t) to denote the impurity function (on a node).

Decision Trees

Decision trees are as you may guess from their name are a machine learning model rooted :-) in an underlying tree-based data structure. Introduced in the 60's and 70's by Morgan/Sondquist (AID/Regression Trees) and Messenger/Mandell (THAID/Classification Trees) they became popular in the

80's and 90's a large part due to the work of Breiman (CART) and Quinlan ($ID_3, C_{4.5}$). Some of the popularity of decision trees can be ascribed to the following characteristics that make them very nice to work with:

- Decision trees can handle heterogeneous data (including a mix of ordered and categorical data)
- intrinsically implemented feature selection
- Easy interpretability
- Non-parametric

We'll be following in large part the CART formulation of Decision Trees introduced by Breiman, et al. in the 1984 work Classification and Regression Trees.

As we've covered in lecture, decision trees are structured models that rely on the patitionability of the input space (predictors/features) and the output space.

Definition: A **tree** is a graph G = (V, E) in which any two vertices (or nodes) are connected by exactly one path. We denote by a **rooted tree** a directed tree in which one of the nodes is designated as the root and all edges are directed away from the root

Definition: In a rooted tree if there exists an edge from v_1 to v_2 (i.e. $(v_1, v_2) \in E$) then v_1 is called the parent and v_2 is called the child. A node with no children is called a **leaf node**. A node that is a parent is called an **internal node**.

Decision trees consist of a root node, internal nodes, and leaf nodes. Each node represents a subspace of X. The internal nodes serve as decisions on some predictor predicate the result of which chooses a child node. Leaf nodes have class labels for the data points in their subspace. The intuition is that we can follow a "divide and conquer approach" to navigate the tree and estimate class labels on X according to the structure of the tree and its induced partition on X.

Definition: A **decision tree** can be described as a model $\phi_{\mathcal{L}} : X \to y$ defined by a rooted tree where any node t represents a subspace $X_t \subset X$ of the input space, with the root node t_0 representing X. Each internal node t is labelled by a split s_t that divides the space X_t that the node t represents into disjoint subspaces corresponding to each of the child nodes.

Definition: Let X_t be the subset of X belonging to node t. A **split** s of node t is a partition of X_t — a set $\{C_i\}$ of non-empty subsets of X_t such that every element $x \in X_t$ is in exactly one of those subsets i.e. $X_t = \sqcup_i C_i$. Furthermore node t is a parent to child nodes t_i with each C_i the subset of X

belonging to node t_i.

The process of learning a decision tree is essentially reconstructing the partition of the input space closest to the partition induced by y on X. Intuitively this involves repeatedly finding the best pair of predictor and split that results in the best partition. But how do we determine the best partition? It turns out that we prefer partitions that result in homogenous class labels. Braiman introduced the idea of an **impurity measure** $i(t)$. An impurity measure is a function on the relative frequencies of classes in the node that gives a value for the (im)purity of the node. While there can be a wide variety of functions that can constitute a good impurity measure, we would expect that any such function to have the following properties:

- The value of i(t) should take a maximum when the class label observations are distributed evenly over all the classes in the node i.e. $(\frac{1}{k}, \frac{1}{k}, \ldots, \frac{1}{k})$
- The value of i(t) should take a minimum when the class label observations belong to a single class i.e. $(1, 0, \ldots, 0)$, $(0, 1, \ldots, 0)$, $\cdots$ $(0, 0, \ldots, 1)$
- $i(t)$ is a symmetric function of the relative class frequencies i.e. $i(t) = \lambda(p_1, p_2, \ldots, p_k)$ and λ is symmetric.

While a number of functions fill these requirements, once we've picked an impurity measure we have all the needed tools to determine how a split at an internal node of the decision tree can affect impurity. For convenience we'll consider binary splits only.

We don't actually know the underlying probability distribution induced by the decision tree but we can approximate it using the proportion of data points in the partition region of that node. So in general if we define p(t) as the probability of being at node t, N_t as the number of samples in the subspace partitioned by node t and the total number of data points as N then $p(t) \approx \frac{N_t}{N}$.

We can also define an aggregate impurity measure for the entire tree as follows.

Definition: The weighted impurity measure for tree T (and its corresponding set of leaf nodes $\tilde{T}$) is

$$I(T) = \sum_{t \in \tilde{T}} i(t)p(t)$$

If we set $I(t) \equiv i(t)p(t)$ i.e. the weighted impurity measure for the node then the above definition has translates well $I(T) = \sum_{t \in \tilde{T}} \text{I(t)}$.

Definition: The "**goodness of split**" or **impurity decrease** $\Delta i(s,t)$ of a binary split dividing node t into a left node t_L and a right node t_R is

$$\Delta i(s,t) = i(t) - p_L i(t_L) - p_R i(t_R)$$

where $p_L \approx \frac{N_{t_L}}{N_t}, p_R \approx \frac{N_{t_R}}{N_t}$ N_{t_L} is the number of samples in the subspace partitioned by the left child node , N_{t_R} is the number of samples in the subspace partitioned by the right child node .

Definition: The **weighted impurity decrease** $\Delta I(s,t)$ is defined as

$$\Delta I(s,t) = p(t)\Delta i(s,t)$$

The **best split** for a node t can be defined by the split that brings the highest impurity decrease. So we iterate through all the possible predictors, find the split on each predictor j that incurs the highes impurity decrease and choose the highest split from that set i.e.

$$s^* = \underset{s_j^*}{argmax} \Delta i(s_j^*, t)$$

Resubstitution Error

The most obvious choice of impurity measure is just what we call the **resubstitution error** or $R(t)$. In classification trees the resubstitution error is also just the classification error — the number of incorrectly classified points in the supspace if we assign every to every point in the subspace the label of the majority of points.

$$i_R(t) = 1 - \underset{k}{max}\ p(k|t)$$

It turns out that R(t) is not a good impurity function in that maximimizing the impurity decrease for R(t) as an impurity measure will not always result in a preference for pure child nodes. So while the properties we laid out for impurity functions form the minimum requirements of an impurity function we need to add an additonal property **strict convexity** are not sufficient to result in functions that reflect our i out that strictly concave impurity functions result in

Some Standard Impurity Functions

In addition to the misclassification rate, we've seen two other common impurity measures in lecture — Entropy and Gini-index. Given a set of class probabilities {p($class_j|t$} at a node t which we'll abbreviate as {p_j}:

Entropy: $i_E(t) = -\sum_j p_j log(p_j)$

Gini-index; $i_G(t) = \sum_j p_j(1 - p_j)$

Stopping Conditions

We grow decision trees according to a greedy process. Stopping conditions vary according to the exact tree modeling variant but the most common conditions include:

- A leaf node is pure (i.e. all the elements in the region partitioned by the leaf node belong to the same class)
- All input variables in a leaf node have the same values (e.g. if you have mislabeling)
- The number of elements in a region falls below a certain minimal threshold
- The max-depth of the tree exceeds a threshold
- The purity decrease $\Delta i(s, t)$ falls below a certain minimal threshold

Pruning

As long as we use $R(T)$ or resubstitution error as the error metric for a tree, the greedy process by which trees are grown means that they are prone to overfitting even with the commonly used stopping conditions.

In particular under classification/resubstitution error we never pay a penalty for splitting a node. If t is the parent node and t_L and t_R the child nodes R(t) > R(t_L) + R(t_R).

Two approaches to controlling overfitting in decision trees are *pre-pruning* and *post-pruning*. Pre-pruning is a set of techniques whereby overfitting is avoided by stopping tree-growth early. Post-pruning relies on growing trees that are expected to overfit and then removing subtrees in order to get optimal trees. We consider only post-pruning methods (which we'll just refer to as pruning from now on) and in particular **minimal cost-complexity pruning**.

Let's take an approach similar to regularization and replace R(T) the resubstitution error on the whole tree by a different cost-complexity function that adds a penalty for complex trees.

Define $|\tilde{T}|$ to be the total number of leaf nodes on a tree T which is a sub-tree of T_{max}. Let's define a new error function indexed by a complexity parameter α.

$$C_\alpha(T) = R(T) + \alpha|\tilde{T}|$$

T_{max} is defined to be tree of maximal size (large enough that we know it will overfit) that we want to prune. T is a subtree of T_{max}. We use $|\tilde{T}|$ as a stand-in for the complexity of the subtree T since the more leaf-nodes a tree has, the more flexibility it has in fitting the training data. C_α looks a lot like the equations for regularization that we've used to combat overfitting in other machine learning methods. Since we aim to replace the classification/resubstitution error R(T) with $C_\alpha(T)$ then we end up penalizing complexity in the form of the total number of leaf nodes in the tree.

From the alpha dependence of C_α we see that as $\alpha \to 0, C_\alpha \to R$. and the penalty for tree complexity goes away removing the incentive to prune. Similarly as $\alpha \to \infty$ then the pruning process will choose the tree composed solely of the root node.

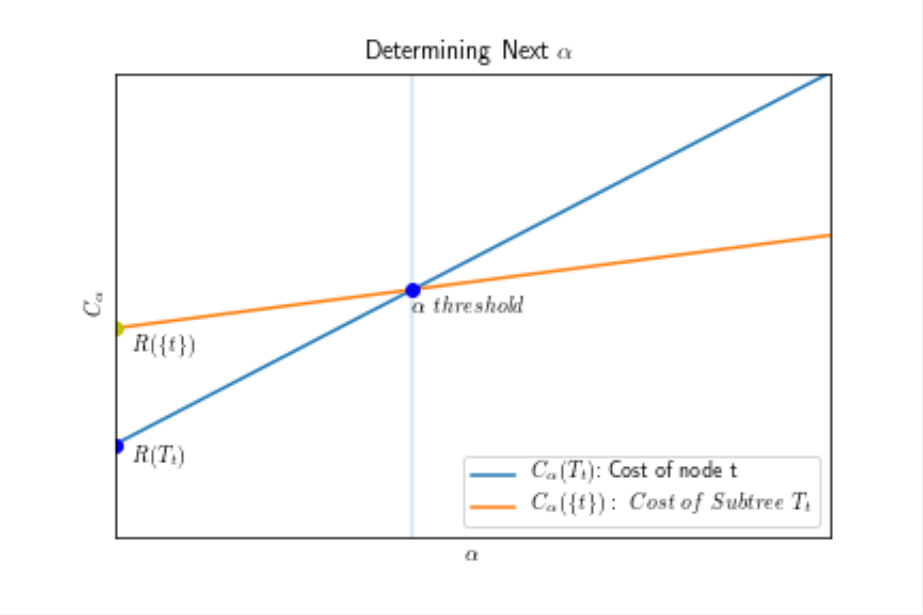
Given a particular α , our goal is to find a $T \subset T_{max}$ that minimizes C_α. Since there are a finite number of subtrees of T_{max}, the existence of a minimizing subtree is guaranteed. In fact, the finite number of subtrees of T_{max} along with the continuous nature of α allow us find the "smallest" subtree T_α that minimizes C_α i.e.

$$1.\ C_\alpha(T_\alpha) = \min_{T \subset T_{max}} C_\alpha(T)$$
$$2.\ C_\alpha(T) = C_\alpha(T_\alpha) \implies T \subseteq T_\alpha$$

There being a finite number of subtrees and thus values of $C_\alpha(T_\alpha)$ means T_α should remain the "smallest" minimum subtree over some interval until at some larger α a new (and smaller) subtree will over take it. How do we find T_α and how does our search for T_α lead us to a procedure for finding optimal pruned subtrees in general. The plan is to generate a telescoping process to construct a set of decreasing subtrees $\{T_1, T_2, \ldots, T_k\}$ along with an accompanying set of increasing α's $\{\alpha_1, \alpha_2, \ldots, \alpha_k\}$ such that for any i>j $T_i \supseteq T_j$ and $\alpha_i > \alpha_j$.

The first step in the process is to determine our starting points α_1 and T_1. We want to set T_1 to the minimum subtree of T_{max} s.t. $R(T_{max}) = R(T_1)$. We can find T_1 by recursively removing all leaf node pairs that don't have a net effect on the R(T) until there are no more such leaf node pairs. We set $\alpha_1 = 0$.

Now that we've got an $\{\alpha_i, T_i\}$ pair how do we find the next pair in the sequence? We



Let's consider cost of replacing a branch of the subtree T_1 with one leaf node — let's call it t. Our new error function C_α applies perfectly well to a subtree consisting of one node. Since the number of leaf nodes $|\tilde{T}|$ for T is set to the node t is just one, then

$$C_\alpha(T) = C_\alpha(t) = R(t) + \alpha|\tilde{t}| = R(t) + \alpha$$

Where $|\tilde{t}|$ is the number of leaf nodes in t which is equal to 1. Similarly the error under our penalized cost function C_α for T_t the branch of subtree T_1 with t as its root node that we're trying to replace with t is similarly defined.

$$C_\alpha(T_t) = R(T_t) + \alpha|\tilde{T_t}|$$

Clearly for α small, we have $C_\alpha(T_t) \approx R(T_t) < R(t) \approx C_\alpha(t)$ but at some threshold α the penalizing effect kicks in and $C_\alpha(T_t) > C_\alpha(t)$. At that threshold α we should replace the branch T_t with the leaf node t. How do we calculate that threshold α? Well it's the α for which the two cost functions match.

$$R(T_t) + \alpha|\tilde{T_t}| = R(t) + \alpha$$
$$\alpha(|\tilde{T_t}| - 1) = R(t) - R(T_t)$$
$$\implies \alpha = \frac{R(t) - R(T_t)}{|\tilde{T_t}| - 1}$$

So now we have a procedure. Given a current tree T_i and a candidate branch T_t to prune and replace with a node t, we calculate the new value of α as above. In order to pick the right value of α_{i+1} and T_{i+1} we calculate candidate alphas for all internal nodes in T_i and pick the node(s) with the smallest α. These nodes are called the "weakest links". Prune the branches of the tree T_t for which these nodes are root and the resulting tree is T_{i+1} and the chosen α from the weakest links becomes α_{i+1}. Rinse and repeat until the final tree which will be the root node of T_{max}. By this process we will have constructed the promised $\{T_1, T_2, \ldots, T_k\}$ and corresponding $\{\alpha_1, \alpha_2, \ldots \alpha_k\}$ with $T_i \subset T_j,\ \alpha_i > \alpha_j\ \forall\ i, j\ s.t.\ i > j$.

Algorithm: Find T_1

- Set $T_1 \leftarrow T_{max}$
    - Choose every parent node t whose two children (t_L and t_R) are leaf nodes
    - Denote this grouping of {t, t_L, t_R} by T_t
    - If R(t) = R(t_L) + R(t_R) prune T_t
    - Set $T_1 \leftarrow T_1 - T_t$
    - Repeat until there are no more candidates T_t for which R(t) = R(t_L) + R(t_R)

Algorithm: Generate $\{T_1, T_2, \ldots, T_k\}$

- Find T_1 from T_{max}
- Set α_1 = 0, k = 1
- While $T_k \supset$ root node(T_{max})
    - Set for every internal node $t \in T_k$: $g_k(t) \leftarrow \frac{R(t) - R(T_t)}{|\tilde{T_t}| - 1}$
    - $\alpha_{k+1} \leftarrow \underset{t \in T_k}{min}\ g_k(t)$
    - Prune the branches of the grouping of nodes for which $g_k(t) == \alpha_{k+1}$
    - increment k

Once we have the sequence $\{T_i\}$ and $\{\alpha_i\}$ we can use one of two methods to pick the right prune size. We can use the full training data set to construct $\{T_i\}$ and then calculate the classification error of each of the T_i on the test set. We pick the model with the lowest error (or in practice as noted by Breiman) we choose the T_i with the smallest α_i with classification performance within 1 SE from the optimal performer as our model.

Another alternative is to use cross validation. We grow T_{max} and generate the $\{T_i\}$ on the entire training set and then set them aside. We employ the usual methods of LOO cross-validations and use the training folds to grow T^f_{max} and generate $\{T^f_i\}$ for each training fold with one caveat. When we generate T_i, α_i is merely the smallest alpha for which T_i is the minimum subtree. In actuality T_i is minimum for any $\alpha \in [a_i, a_{i+1}]$. When generating $\{T_i\}$ in each fold, we instead of α_i we use $\sqrt{\alpha_i \alpha_{i+1}}$ as a "more typical" value of α for T_i. After training and testing the generated sequences $\{T_i\}$ on each fold we choose the T_i with the lowest CV classification error and then choose as our model the corresponding T_i grown on the entire training set.

Handling Missing Data in Decision Trees

We've considered the problem of missing data in relation to previous machine learning and statistical techniques. In general, tree based models assume access to the complete set of observations and as we've seen use those observations to generate the tree — both the decision questions for each node as well as the assignment of each data point to the correct partition.

In many situations — in particular if the "missingness" is unconnected to the observed predictors — even a small probabilty of a predictor being missing (5-10%) within an observation can have a parachuting effect on the total number of data points with missing observations. One common strategy for dealing with missing data, dropping all the data points with missing predictors, can be problematic because the sheer volume of observations being discarded decreases the potential efficacy of the trained decision tree model. In addition, if the missing attributes aren't missing at random, then discarding the data with missing attributes can potentially introduce bias into our analysis.

It's still possible to deal with the missing data by means of mean, median and/or model based imputation, but we'd like to introduce a strategy (originated by Breiman 1984) for missing data based on properties intrinsic to tree based models and in particular **CART**. Our strategy needs to address two major issues:

- How does it determine the best split if some of the predictors have missing observations?
- How does it assign a data point to the appropriate node if the data point is missing a value for the predictor that gives the optimal split?

Recall that in determining the best split **CART** finds the split s for node t that optimizes the "goodness of split"

$$\Delta i(s,t) = i(t) - p_L i(t_L) - p_R i(t_R)$$

In the case of missing data, we can compute this optimal best split while ignoring any missing values. In practice this means that for each split (determined for a particular predictor), you use only data points with observed values of that predictor to calculate $\Delta i(s,t)$.

Assigning a data point with a missing value for the decision variable at a node is more problematic. We'll take advantage of a notion introduced by Breiman called *surrogate splits*. In finding the best split s^* at node t, one really optimizes over splits s for that node and as a side affect find the variable X_i giving the optimal split. Once the best split is found, a set of surrogate splits $\{\tilde{s_j}\}$ is generated where the index j runs over all the variables not selected by the optimal split. For each variable X_j we find the split that matches most closely with the best split s^*. We denote that split as $\tilde{s_j}$ and add it to the set of surrogate splits.

To clarify what we mean by a closer match, let's look at an example. Denote the example data points as {a, b, c, d, .., g}. Let's assume thtat s^* induces a split of {a, b, c, d} to the left child node and {e, f, g} to the right child node. If we have two candidate surrogate splits s_1 and s_2 with s_1 inducing a split {a, d, e} to the left child node and {b, c, f, g} to the right then we assign s_1 a similarity score of 4/8 or 0.5 since it assigns half of its elements to the same child nodes that s^* does. Similarly if s_2 induces a split of {a, b, d} to the left child node and {c, e, f, g} it gets a similarity score of 7/8 or .875. s_2 then is a closer match to $s*$ and would rank ahead of s_1 as surrogate split.

We sort the set $\{\tilde{s_j}\}$ according to similarity score. The surrogate split with the highest score is defined to be the **best surrogate split**. When trying to assign a data point with a missing value on the decision variable in the node, we use the surrogate splits instead assigning the data point according to the first split in $\{\tilde{s_j}\}$ for which the point has a value for that predictor.

Variable Importance

It's possible for a dataset to contain a very high number of predictors (sometimes with the number of predictors far exceeding the number of data points). Variable importance is a model specific measure of which features have the largest predictive influence on the model. In some analyses (especially those dealing with an overabundance of predictors) feature selection is an important goal and a model that lends itself towards easily interpretable variable importance is attractive. Decision trees and their related ensemble models (Gradient Boosted Trees and Random Forests) have related notions of variable importance based on which variables show up in

the decision predicates in Decision Trees.

For an individual decision tree T a natural approach is to measure for each variable its impurity decrease contribution over all best splits

$$Imp(X_i) = \sum_{t \in T} 1(var_{s_t} = X_i) \Delta I(s_t^*, t)$$

where we denote with the indicator function $1(var_{s_t} = X_i)$ that the split is taking place over the variable X_i. Breiman pointed out some problems related to this approach, primary among them variable masking. Consider variables X_{m_1} and X_{m_2}. X_{m_2} may be masked by X_{m_1} i.e. if at various nodes X_{m_1} generates the best split, it turns out that on these nodes X_{m_2} generates a split that's almost as good. Since X_{m_2} never generates a best split though it will be given a low variable importance even though were X_{m_1} removed, X_{m_2} would fill in without notable effect on the accuracy, etc. To counter variable masking, Breiman proposed aggregating the impurity decrease every time a variable shows up as the best split or the best surrogate split.

$$Imp(X_i) = \sum_{t \in T} \Delta I(\tilde{s}_t^i, t)$$

where $\tilde{s}_t^i$ denotes best primary or surrogate split.

In Random Forests, variable masking issues are assuaged to a large degree by random variable subselection. If X_{m_2} is masked by X_{m_1} it can still make a contribution on trees for which X_{m_1} hasn't been selected. So we can aggregate over all the trees in the forest the impurity decrease contribution over best splits without the variable masking concerns. This aggregate value is known as the **Mean Decrease Genie** and is one of two common variable importance mechanisms in Random Forest. If the Random Forest has M trees we can write **MDG** as follows:

$$Imp(X_i) = \frac{1}{M} \sum_{m} \sum_{t \in T_m} 1(var_{s_t} = X_i) \Delta I(s_t^*, t)$$

where T_m is the m^{th} tree and the indicator function is the same one defined above.

Another measure of variable importance in Random Forests is called **Mean Decrease Accuracy**. If we're trying to measure the importance of variable X_i we train the random forest model, calculate the accuracy of model, permute the OOB values of variable X_i, and then calculate the accuracy of the model again. If X_i is important then the shuffling of its OOB values will have a deleterious impact on the accuracy of the model. In practice this process takes place via cross validation to get a stable estimate.

Algorithm: Find MDA

- Create Cross Validation Folds. For each Fold:
    - Fit Random Forest model
    - Calculate the accuracy of the model
    - Choose a variable X_i
        - copy OOB set
        - Permute the values for X_i in OOB set
        - measure the accuracy of the model
        - $mda_i \leftarrow -\Delta$ in accuracy
        - repeat for all the X_i
    - Repeat for all the folds

Since Boosted Gradient Trees are a weighted ensemble of decision trees the variable importances of the decision trees are weighted and aggregated to generate the variable importances for the gradient boosted tree model.