CS229 Lecture Notes: Decision Trees

Selwin George

Latest revision: 26 August 2023

These notes are adapted primarily from [Mur22] and [SB14].

1 Decision Trees

1.1 Definition

Formally, a **decision tree** can be thought of as a mapping from some k regions of the input domain $\{R_1, R_2, \ldots, R_k\}$ to k corresponding predictions $\{w_1, w_2, \ldots, w_k\}$. We require these regions to partition the input domain, meaning that there is no intersection between any two regions, and the union of the regions recovers the entire input domain. We also require that our prediction for any point in a certain region R_j is the same, namely w_j. For example, we could create a decision tree where our regions $R_1, \ldots, R_4$ are the four quadrants in $\mathbb{R}^2$, i.e.,

$$\begin{aligned} R_1 &= \{x : x_1 \geq 0, x_2 \geq 0\}, \\ R_2 &= \{x : x_1 < 0, x_2 \geq 0\}, \\ R_3 &= \{x : x_1 < 0, x_2 < 0\}, \\ R_4 &= \{x : x_1 \geq 0, x_2 < 0\}, \end{aligned}$$

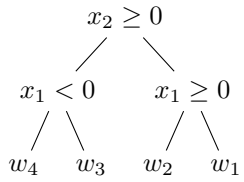
and we predict some w_1 for all $x \in R_1$, w_2 for all $x \in R_2$, and so on.

More generally, we can express our prediction on a given input x using a fairly compact algebraic representation:

$$f(x) = \sum_{j=1}^{k} w_j \mathbf{1}[x \in R_j].$$

Note that the terms in this summation over the k regions will all be 0 except for that of the region R_j in which x is located. In the regression case, taking $w_j = \frac{\sum_{i=1}^{n} y^{(i)} \mathbf{1}[x^{(i)} \in R_j]}{\sum_{i=1}^{n} \mathbf{1}[x^{(i)} \in R_j]}$, the average value of the labels of the training data in region R_j, is the common choice (but not required). For classification, it is typical to let w_j instead be whichever class is most common among the training data in R_j (a "plurality vote" within the relevant region).

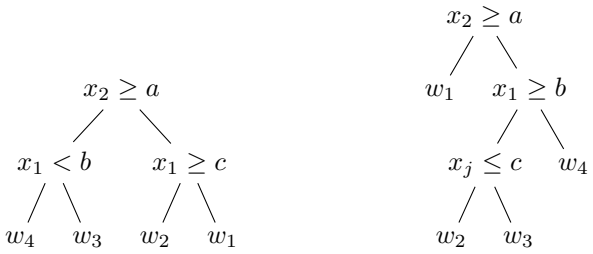
Furthermore, this kind of mapping can be easily represented as a tree, hence the name. Using the example above of splitting $\mathbb{R}^2$ into quadrants, we can structure a corresponding tree as



where we follow the the right branch of a node if the condition is met, and the left one if not. We can notice here that each of our regions R_j corresponds to a single leaf (childless node) in this tree representation; indeed, this property will hold for all decision tree models we study, meaning that the number of regions k in a decision tree model is exactly the number of leaves in any corresponding tree representation.

In these notes we will be assuming we have a training set containing n datapoints, each being d-dimensional. Therefore, each of our regions will be a subset of $\mathbb{R}^d$, and we'll assume that each of our corresponding predictions is a real number. Additionally, we will be following the standard practice of requiring each tree node's branching condition to be a single inequality on a single feature, e.g., $\mathbf{1}[x_j < s]$ or $\mathbf{1}[x_j \geq t]$ where $s, t \in \mathbb{R}$ and $j \in \{1, \ldots, d\}$. The result of following this practice is that each of our regions will be an axis-aligned subset of $\mathbb{R}^d$, meaning each edge of a region will be parallel to one axis and perpendicular to the rest.

Here's an exercise to check your understanding before moving on. Consider the following two decision tree models where $d = 2$, $a, b, c \in \mathbb{R}$, and $j \in \{1, 2\}$:



For each of these models, what (if any) are the restrictions on a, b, c and j if we require that all four predictions $w_1, \ldots, w_4$ are possible? And can you draw out the resulting regions in $\mathbb{R}^2$ along with their corresponding predictions for a fixed choice of (suitable) a, b, c and j?

1.2 Model fitting

To fit the model, we seek to optimize a loss function just as we would in other supervised learning models. Here, our loss function looks like

$$\mathcal{L}(f) = \sum_{i=1}^{n} \ell\big(y^{(i)}, f(x^{(i)})\big) = \sum_{j=1}^{k} \sum_{x^{(i)} \in R_j} \ell(y^{(i)}, w_j).$$

The first equality is a general form familiar to us from our study of other supervised learning models, while the second gives an equivalent representation using the specifics of the decision tree model.

But unlike with the cases we've seen before, this is not a smooth loss function due to the discrete structure of the decision tree itself; the tools of calculus that we've been using so far in supervised learning cannot help us with loss minimization here. It turns out that calculating an optimal (maximally efficient) tree model to minimize this loss function is NP-complete, i.e., intractable as far as we know. Instead, we use a greedy algorithm to fit our model to the data. In our description of the training algorithm in this section, we'll refer to an abstract "gain" function $G(\cdot, \cdot)$ and assume that this function gives us some measure of the improvement that comes from doing a certain split at a given node. We'll make this function more concrete shortly.

The algorithm is recursive at each node of the tree. We start with a single node that will form the root of our tree, and we'll say the node carries the full training set. Now to apply the algorithm at a given node, we decide whether or not to split the node according to some condition $\mathbf{1}[x_j \geq \theta]$ for some $j \in \{1, \ldots, d\}$ and some $\theta \in \mathbb{R}$.

Let's first assume we want to split the node. To choose our split, we choose

$$j, \theta = \arg\max_{j,\theta} G(j, \theta).$$

In words, we find the feature j and the split value θ such that we maximize our gain function $G(j, \theta)$, which again is just an abstract placeholder for the improvement that comes from this particular split. We then partition the data accordingly into the left and right children that result from this split, and we carry out the same recursion afterwards on each child.

If we choose not to split the node (we will discuss when we make this choice shortly), then we assign a prediction w for the region corresponding to this (leaf) node, usually according to the heuristics described in the previous section.

Now, the optimization $j, \theta = \arg\max_{j,\theta} G(j, \theta)$ that we use to choose a split might seem intractable at first glance. We can certainly iterate over the finite possibilities for j, but this kind of iteration looks unavailable to us for $\theta \in \mathbb{R}$. However, we'll now see that there are actually only finitely many "distinct" values of θ in terms of the output of our gain function $G(\cdot, \cdot)$.

Consider a certain feature j. Then we can list out feature j for each datapoint as $x_j^{(1)}, \ldots, x_j^{(n)}$. We can sort the datapoints by this feature value such that $x_j^{(i_1)} \leq x_j^{(i_2)} \leq \cdots \leq x_j^{(i_n)}$ where $i_1, \ldots, i_n$ gives the ordering of the datapoints

in ascending value of their jth feature. Here, we see that choosing any θ between two of these values is functionally equivalent in terms of splitting the data in the node—all datapoints less than θ in this chain of inequalities get sent into the left child, and all that are greater get sent to the right. So, there are functionally $n-1$ choices of θ, and it is typical to choose $\theta = \frac{x_j^{(i_\ell)} + x_j^{(i_{\ell+1})}}{2}$ when seeking the split between the ℓth and $(\ell+1)$th values in the sorted list. Therefore, we can find j and θ as described above by iterating over the $d \times (n-1)$ possible distinct splits available to us.

1.3 Maximizing gain by minimizing cost

We now concretize our notion of the gain function $G(\cdot, \cdot)$ above. But first, let's introduce yet another abstract function $C(\cdot)$ to represent the cost associated with the datapoints associated with a certain node. Let S be the set of datapoints associated with the node whose split we wish to evaluate. Let L and R be the datapoints associated with the left and right children of the original node after splitting according to $\mathbf{1}[x_j \geq \theta]$. An intuitive representation of our gain function in terms of our cost function would be

$$G(j, \theta) = C(S) - \left[\frac{|L|}{|S|}C(L) + \frac{|R|}{|S|}C(R)\right]$$

since this is the difference between the cost before splitting $C(S)$ and the (proportionally weighted) cost after splitting $\frac{|L|}{|S|}C(L) + \frac{|R|}{|S|}C(R)$. It follows that choosing a split that maximizes the gain function is equivalent to choosing a split that minimizes the cost after splitting (can you see why?).

To fully concretize the gain function, all that remains now is to make our cost function explicit. There are many usable cost functions $C(\cdot)$, perhaps most straightforward of which would be our loss function

$$\frac{1}{|S|} \sum_{x^{(i)} \in R} \ell(y^{(i)}, w)$$

where S is the set of datapoints associated with the node whose cost we evaluate, R is the region associated with this node, and w is the prediction for region R. But reusing the loss function isn't necessary here—one viable cost function for a regression tree would be the mean squared error

$$\frac{1}{|S|} \sum_{x^{(i)} \in R} (y^{(i)} - w)^2,$$

and one for the classification case would be the entropy

$$\sum_{c \in \mathcal{C}} -p_c \log p_c$$

where $\mathcal{C}$ is the set of classes and p_c is the fraction of datapoints in S of class c.

1.4 Regularization

How well can a tree do on an arbitrary modeling task? In other words, if we let the greedy algorithm described above run indefinitely, what happens?

A tree of sufficient depth can perfectly fit (achieve 0 loss) on any internally-consistent training set. Furthermore, our greedy algorithm will achieve this fit eventually (can you see why?), though usually suboptimally as noted earlier. However, allowing the tree to reach such depths often invites the twin problems of overfitting to our training set and creating an undesirably large model. For the sake of regularization, heuristics are typically used for deciding not to split a node further. A few of these heuristics are limiting overall tree depth, limiting the number of leaves (distinct regions) in the model, barring additional splits when there are too few datapoints in a node, and not splitting if the gain described above is beneath some threshold.

Another method for regularization would be to "prune" the tree after letting it (greedily) reach a 0-loss fit. **Pruning** generally involves creating another function that evaluates whether the increased performance associated with a node (or set of nodes) is worth the extra model size and reduced generalization. If not, one can "prune" the node (nodes) by merging it (them) back into tree. Pruning can often be done greedily, as was our fitting algorithm, but starting from leaves instead of the root of the tree.

1.5 Summary

1.5.1 Positives

Advantages of using trees include but are not limited to:

- Highly interpretable
- Robust to outliers
- Robust to mix of continuous and discrete features
- Robust to monotone transformations of input (transformations that don't alter the output of our sorting strategy from above)
- Can usually achieve a "decent" fit relatively quickly, even on large datasets

1.5.2 Negatives

Disadvantages of using trees include but are not limited to:

- Tend to generalize poorly, even when regularized
- Highly unstable around boundaries

2 Improvements via Ensemble Learning

Ensemble learning is method in machine learning that reduces model instability by averaging predictions over multiple instances of similar models. Given some individual models $F_1, \ldots, F_m$ (not necessarily decision trees), we can write our overall ensemble model output as

$$f(x) = \sum_{i=1}^{m} \beta_i F_i(x),$$

where β_i is a weight for each model's prediction. It is often the case that $\beta_1, \ldots, \beta_m$ are all taken to be $\frac{1}{m}$ to imply no favoritism between models. Ensemble learning tends to reduce variance without significantly increasing bias, resulting in improved performance overall. A downside is, of course, the computational cost of training and using additional models.

This kind of technique is relevant in both the regression and classification cases. In regression, the averaging mentioned above can be interpreted literally, whereas in classification we can use a plurality vote again, this time among the individual classifier predictions, to decide the overall ensemble classification.

2.1 Bagging

The first ensemble method we'll introduce is **bagging**, which stands for "bootstrap aggregating". **Bootstrapping** is a general technique where we aim to simulate drawing a new sample from the true underlying distribution from which our training set is generated (since we don't have access to the true distribution outright). In this practice, we sample from our training set as if the training set itself is a perfect representative of the true distribution. Concretely, we generate a "new" dataset by sampling uniformly and with replacement from our original one. The aggregating component of bagging is to apply this bootstrapping procedure once for each model in our ensemble so that each one can be trained in parallel on a unique dataset, usually of the same size n as the original dataset.

To see why this kind of intervention is helpful, consider what would happen if we simply trained each model in our ensemble on the same dataset. If our training algorithm is deterministic, like our tree fitting algorithm described above, then there will be no differences between the outputs of the individual models, meaning we functionally haven't done any ensemble learning (we've only wasted computational resources). But how different are the underlying models that result from bagging if they're still each ultimately trained on datapoints from the same dataset?

We can think somewhat concretely about this in the case where each of our sampled datasets is of the same size as the original, as is customary in bagging. Consider a single bootstrapped sample of size n. For each datapoint in the original dataset, we have that with probability $(1 - \frac{1}{n})$ that datapoint goes unselected if we choose a single datapoint uniformly at random. Also, since the datapoint has equal and independent probability of being chosen at any of the

n in our sample, we can write the probability of that datapoint appearing at least once in our sample as $1 - \left(1 - \frac{1}{n}\right)^n$. Recognize that in the limit we have

$$\left(1 - \lim_{n \to \infty} \left(1 - \frac{1}{n}\right)^n\right) = 1 - e^{-1} \approx 0.632.$$

So, in expectation, only 63.2% of the original datapoints appear at all in a sample of size n when n gets large, informally speaking.

So, we have that each individual model will only (on average) see 63.2% of the data, which is disadvantageous in terms of individual models' performances on the overall dataset, but advantageous in the sense that it prevents the ensemble as a whole from fixating too much on a few datapoints, lowering the overall variance.

But is bagging sufficient for creating significant variation between the models in our ensemble? This largely depends on the underlying models themselves. If the models are sufficiently low variance (which as we've learned often implies having high bias), then bagging is likely to produce highly similar models, reducing the value of this intervention. However, if we use unstable (high variance) models, like decision trees, then we are effectively harnessing the instability of our base learner to help ensure the quality of our ensemble learning procedure.

2.1.1 Random Forests

The **random forests** algorithm involves taking a step beyond bagging when we have decision trees as our base learners. To further decrease the correlation between individual models, in each of our individual trees, we'll consider only a subset of features at each node-splitting step in our greedy algorithm. Often this subset will be chosen uniformly randomly at each node, where the subset size is fixed to some hyperparameter $k < d$. Notice that this practice mirrors that of bagging in the following sense: in bagging, each individual model has access to any of the original datapoints in theory—we aren't brute-force barring any datapoints from appearing in any of our bootstrapped samples. Similarly, we aren't removing any features from the datapoints of individual trees in a random forest—we are simply adding randomness to their availability at each node-splitting step to encourage diversity among our base learners. In practice, random forests tends to outperform bagging alone when decision trees are the base learner.

2.2 Boosting

Stepping back from trees again, recall our expression for the ensemble model

$$f(x) = \sum_{i=1}^{m} \beta_i F_i(x).$$

As mentioned, our overall goal with this (or any) model in the supervised setting is to minimize a loss function of the form

$$\mathcal{L}(f) = \sum_{i=1}^{n} \ell\big(y^{(i)}, f(x^{(i)})\big).$$

In contrast to bagging, where a key factor was the fact that each individual model in the ensemble can be trained in parallel, another technique called **boosting** involves, at a high level, sequentially developing the next individual model F_{i+1} based on the performance of f_i, where we take f_i to be the ensemble model

$$f_i(x) = \sum_{j=1}^{i} \beta_j F_j(x).$$

Boosting was originally created specifically for the binary classification case, but the core idea has since been generalized to other prediction tasks. The origins of boosting were closely tied to theoretical ideas of weak and strong learners, which we'll expand on in the next section, but the overall idea is that underpowered models, when combined, can on the whole become capable of an otherwise infeasible prediction task. Thus, a strong learner can be developed from the amalgamation of several easy-to-train and easy-to-interpret weak learners, and sometimes this ensemble of weak learners is preferable to developing a single strong one. In contrast to bagging and random forests, which mostly aim to reduce ensemble variance, boosting aims to reduce model bias via this strengthening process which we'll describe in more detail.

The type of boosting we'll cover is referred to as **forward stagewise additive modeling**, and it classically takes the following form:

At iteration i of our ensemble creation, we create the model F_i. To do so, we compute

$$\beta_i, \theta_i = \arg\min_{\beta,\theta} \mathcal{L}(f_{i-1} + \beta F_i) = \arg\min_{\beta,\theta} \sum_{j=1}^{n} \ell\Big(y^{(j)}, \big(f_{i-1}(x^{(j)}) + \beta F_i(x^{(j)}; \theta)\big)\Big),$$

where θ represents the parameters of the new model F_i. We then set $F_i(x) := F_i(x; \theta_i)$ and afterwards set

$$f_i(x) := f_{i-1}(x) + \beta_i F_i(x)$$

to define our new ensemble model.

However, this optimization implies a parametric form for F_i, where $F_i(\cdot; \theta_i)$ is a specific choice within a model family parameterized by θ. Our treatment of boosting will take a slightly different form that can be generalized to nonparametric models (e.g., decision trees). But first, we'll say a bit about strong and weak learners to give some intuition for boosting.

2.2.1 Weak learners vs. strong learners

Having a vague sense of what we mean by weak learner and strong learner is more important for these notes than a comprehensive formal understanding. But to develop this sense, we'll look at some formalism for the binary classification case. Let $\delta, \epsilon \in (0,1)$, and let $\gamma \in (0, \frac{1}{2})$. These will represent tolerance of uncertainty, tolerance of inaccuracy, and achievable improvement in accuracy, respectively. Roughly speaking, we say a learning algorithm is a **strong learner** when for any choice of δ, ϵ, we have some $m = m_{\mathcal{H}}(\delta, \epsilon) \in \mathbb{N}$ such that, after learning on at least m i.i.d. samples from the true distribution, we have

$$\mathbb{P}\big(\text{Accuracy}(f) > 1 - \epsilon\big) \geq 1 - \delta$$

where f is the model output by the learning algorithm. In words, we're saying there's always some number of training examples that we could see such that we are guaranteed to learn a (probably, approximately) correct model. We'll call a learning algorithm a γ-**weak learner** if for any choice of δ, we have some $m = m_{\mathcal{H}}(\delta) \in \mathbb{N}$ such that, after learning on at least m i.i.d. samples from the true distribution, we have

$$\mathbb{P}\big(\text{Accuracy}(f) > \frac{1}{2} + \gamma\big) \geq 1 - \delta.$$

This looks similar to our statement about strong learners, but it is quite a bit weaker—we are only saying that there is always some number of training examples that we could see such that we learn a model that (probably) does a little bit better than randomly guessing on the binary classification task, which would yield an accuracy of 0.5. Note that in these examples, the Accuracy function is taken to give the expected accuracy of a model with respect to the true distribution.

Again, this is just to give a slightly more concrete sense as to what we mean when invoking the terms weak learner and strong learner, and thereby impart a sense for why additively combining weak learners together to make a strong one is interesting and not obviously feasible a priori.

2.2.2 Gradient boosting

Our treatment of boosting will center around gradient boosting, which allows the generalization to nonparametric models that we alluded to earlier. In **gradient boosting**, instead of learning some parameters β_i, θ_i to develop $F_i(\cdot; \theta_i)$, we instead will be concerned with learning the entire function F_i itself:

$$F_i = \arg\min_F \mathcal{L}(f_{i-1} + F) = \arg\min_F \sum_{j=1}^{n} \ell\Big(y^{(j)}, \big(f_{i-1}(x^{(j)}) + F(x^{(j)})\big)\Big).$$

Notice now that the optimization is done over the space of functions, which is a larger task than just estimating a finite list of parameters θ. Indeed, we do

not attempt to complete this optimization outright. Instead, we carry out this optimization in the form of (functional) gradient descent. At iteration i, let

$$f = \Big(f_{i-1}(x^{(1)}), \ldots, f_{i-1}(x^{(n)}) \Big)$$

be the vector of our current function's evaluations on our datapoints. Additionally, let

$$g = \left(\frac{\partial \ell(f_{i-1}(x^{(1)}), y^{(1)})}{\partial f_{i-1}(x^{(1)})}, \ldots, \frac{\partial \ell(f_{i-1}(x^{(n)}), y^{(n)})}{\partial f_{i-1}(x^{(n)})} \right)$$

be the gradient of our loss function with respect to f. Then we see that $f := f - \alpha g$ can be thought of as an updated vector of evaluations after a single step of gradient descent, where α is our learning rate.

However, this alone isn't too useful to us. We don't just seek to update f, which is our model's outputs on our training datapoints; we want to update f_{i-1} for all possible inputs x. So simply letting $f_i := f_{i-1} - \alpha g$ doesn't make sense syntactically or semantically—we'd be taking the difference between a function (which we've seen can be represented as a infinite-dimensional vector) and an n-vector. We'd need the new function F_i to be defined over all real numbers to achieve the function-level gradient descent we're after.

In order to get a more general representation of g that extends to all inputs, we can fit a weak learner to our gradient step via a supervised learning task. In particular, we can write

$$F_i := \arg\min_{F} \sum_{j=1}^{n} \big(-g_j - F(x^{(j)}) \big)^2,$$

which can be solved with a separate training algorithm for whatever model F we choose. The resulting function F_i will be defined for all inputs, as desired, though it will prioritize a fit between each training datapoint $x^{(i)}$ and the negative of the corresponding component of the gradient g. So, we can now write

$$f_i(x) := f_{i-1}(x) + \alpha F_i(x)$$

to accomplish our function-level gradient descent interpretation of boosting.

Note that as we iterate through this process, we'll be fitting a model to the gradient of our loss with respect to our model's current predictions, meaning that each successive model we add is tailored to weaknesses in the current ensemble model. Under this interpretation, we can think of boosting as simply adding on models that are trained according to a relabelled version of the original dataset, where the relabelling is done to improve performance on these aforementioned weaknesses.

Where do trees fit into this? Remember, in the boosting paradigm we want fast, weak learners that in the aggregate can form a powerful ensemble model. Decision trees are considered weak learners when they are highly regularized, and thus are a perfect candidate for this role. In fact, gradient boosting in practice nearly always uses decision trees as the base learner (at time of writing).

So, we can take $F_1, F_2, \ldots$ to each be trees fit according to the iterative procedure above (usually with heavy regularization), and we can stop creating new trees whenever we reach some thresholds on model performance, model size, etc.. Gradient boosting with decision trees as the base learner is called **gradient tree boosting**, but in common usage these terms are often interchanged (at time of writing).

2.2.3 XGBoost

Most if not all of the algorithms and techniques we study in this class do not see real-world use without significant enhancements. This is to be expected given the introductory nature of the course. However, as we conclude our discussion of decision trees, we are actually quite a bit closer to the edge of the field than we've been with the other topics we've covered in the course. With gradient tree boosting in our toolkit, we're nearly able to implement a genuine state-of-the-art algorithm, namely **XGBoost**, which sees out-of-the-box usage in industry settings regularly (at time of writing).

The main differences between XGBoost and gradient tree boosting are that

- We optimize the loss function with a regularization term included:

$$\mathcal{L}(f) = \left[\sum_{i=1}^{n} \ell\big(y^{(i)}, f(x^{(i)})\big)\right] + \gamma J + \frac{1}{2}\lambda \sum_{j=1}^{J} w_j^2.$$

  J is the number of leaves, w_j is the prediction value associated with the jth leaf in the model, and γ, λ are scalar-valued hyperparameters.

- We use a subset of features when splitting tree nodes like in random forests.

- We use a second-order approximation rather than just a gradient, i.e., the Hessian of the loss with respect to our current predictions makes an appearance in our boosting algorithm along with the gradient.

For more details on XGBoost, see https://github.com/dmlc/xgboost.

References

[SB14] Shai Shalev-Shwartz and Shai Ben-David. *Understanding Machine Learning: From Theory to Algorithms*. Cambridge University Press, 2014.

[Mur22] Kevin P. Murphy. *Probabilistic Machine Learning: An introduction*. MIT Press, 2022.