How to tune a Decision Tree?

Mukesh Mithrakumar Nov 11, 2019 · 16 min read

How do the hyperparameters for a decision tree affect your model and how do you choose which ones to tune?

Hyperparameter tuning

> Hyperparameter tuning is searching the hyperparameter space for a set of values that will optimize your model architecture.

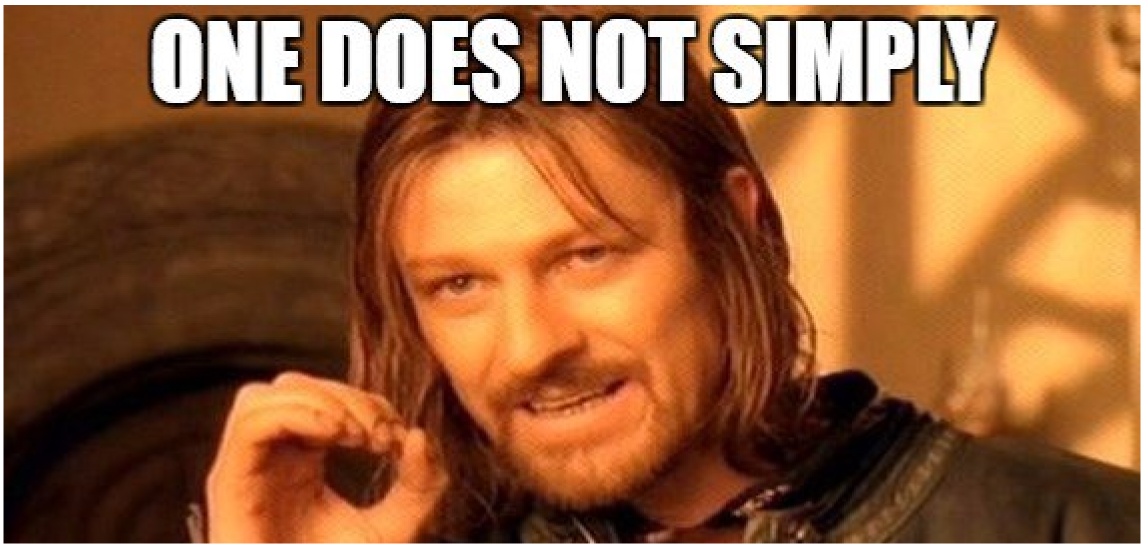
This is different from tuning your model parameters where you search your feature space that will best minimize a cost function.

Hyperparameter tuning is also tricky in the sense that there is no direct way to calculate how a change in the hyperparameter value will reduce the loss of your model, so we usually resort to experimentation. This starts with us specifying a range of possible values for all the hyperparameters. Now, this is where most get stuck, what values am I going to try, and to answer that question, you first need to understand what these hyperparameters mean and how changing a hyperparameter will affect your model architecture, thereby try to understand how your model performance might change.

The next step after you define the range of values is to use a hyperparameter tuning method, there's a bunch, the most common and expensive being Grid Search where others like Random Search and Bayesian Optimization will provide a "smarter", less expensive tuning. These methods are not really the focus of this article but if you want to learn more, check the reference section [1].

Decision Tree

Decision Tree is one of the popular and most widely used Machine Learning Algorithms because of its robustness to noise, tolerance against missing information, handling of irrelevant, redundant predictive attribute values, low computational cost, interpretability, fast run time and robust predictors. I know, that's a lot 😂. But a common question I get asked from students is how to tune a Decision Tree. What should be the range of values I should try for the maximum depth, what should be the minimum number of samples required at a leaf node? These are very good questions that don't have a straightforward answer but what we can do is understand how changing one will affect your model. Like what does increasing the maximum depth really mean, what does changing the minimum sample leaves do to your model. So, in this article, I attempt to give you an introduction to these parameters and how they affect your model architecture and what it can mean to your model in general.



Let's look into Scikit-learn's decision tree implementation and let me explain what each of these hyperparameters is and how it can affect your model. Btw note that I assume you have a basic understanding of decision trees.

Since the decision tree is primarily a classification model, we will be looking into the decision tree classifier.

DecisionTreeClassifier

criterion: string, optional (default="gini"):

> *The function to measure the quality of a split. Supported criteria are "gini" for the Gini impurity and "entropy" for the information gain.*

If you ever wondered how decision tree nodes are split, it is by using impurity. Impurity is a measure of the homogeneity of the labels on a node. There are many ways to implement the impurity measure, two of which scikit-learn has implemented is the Information gain and Gini Impurity or Gini Index.

Information gain uses the entropy measure as the impurity measure and splits a node such that it gives the most amount of information gain. Whereas Gini Impurity measures the divergences between the probability distributions of the target attribute's values and splits a node such that it gives the least amount of impurity.

According to the paper "Theoretical comparison between the Gini Index and Information Gain criteria" [3], the frequency of agreement/disagreement of the Gini Index and the Information Gain was only 2% of all cases, so for all intents and purposes you can pretty much use either, but the only difference is entropy might be a little slower to compute because it requires you to compute a logarithmic function:

$$Gini: Gini(E) = 1 - \sum_{j=1}^{c} p_j^2$$

$$Entropy: H(E) = -\sum_{j=1}^{c} p_j \log p_j$$

Many of the researchers point out that in most of the cases, the choice of splitting criteria will not make much difference in the tree performance. Each criterion is superior in some cases and inferior in others, as the "No Free Lunch" theorem suggests.

splitter: string, optional (default="best")

> *The strategy used to choose the split at each node. Supported strategies are "best" to choose the best split and "random" to choose the best random split.*

According to scikit-learn's "best" and "random" implementation [4], both the "best" splitter and the "random" splitter uses Fisher-Yates-based algorithm to compute a permutation of the features array. You don't really need to worry about the algorithm, the only difference is, in the "best" splitter it evaluate all splits using the criterion before splitting whereas the "random" splitter uses a random uniform function with min_feature_value, max_feature_value and random_state as inputs. We will look into what these are below but for now, let's see how the splitter will affect the model.

Let's say you have hundreds of features, then "best" splitter would be ideal because it will calculate the best features to split based on the impurity measure and use that to split the nodes, whereas if you choose "random" you have a high chance of ending up with features that don't really give you that much information, which would lead to a more deeper less precise tree.

On the other hand, the "random" splitter has some advantages, specifically, since it selects a set of features randomly and splits, it doesn't have the computational overhead of computing the optimal split. Next, it is also less prone to overfitting because you are not essentially calculating the best split before each split and the additional randomness will help you here, so if your model is overfitting, then you can change the splitter to "random" and retrain.

So for a tree with few features without any overfitting, I would go with the "best" splitter to be safe so that you get the best possible model architecture.

max_depth: int or None, optional (default=None)

> *The maximum depth of the tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples.*

The theoretical maximum depth a decision tree can achieve is one less than the number of training samples, but no algorithm will let you reach this point for obvious reasons, one big reason being overfitting. Note here that it is the number of training samples and not the number of features because the data can be split on the same feature multiple times.

Let's first talk about the default None case, if you don't specify a depth for the tree, scikit-learn will expand the nodes until all leaves are pure, meaning the leaf will only have labels if you choose default for the min_samples_leaf, where the default value is one. Note that most of these hyperparameters are tied to one another and we will talk about the min_samples_leaf shortly. On the other hand, if you specify a min_samples_split, which we will look at next, the nodes will be expanded until all leaves contain less than the minimum number of samples. Scikit-learn will pick one over the other depending on which gives the maximum depth for your tree. There's a lot of moving parts here, min_samples_split and min_samples_leaf so let's just take the max_depth in isolation and see what happens to your model when you change it, so after we go through min_samples_split and min_samples_leaf we can get a better intuition of how all these come together.

In general, the deeper you allow your tree to grow, the more complex your model will become because you will have more splits and it captures more information about the data and this is one of the root causes of overfitting in decision trees because your model will fit perfectly for the training data and will not be able to generalize well on test set. So, if your model is overfitting, reducing the number for max_depth is one way to combat overfitting.

It is also bad to have a very low depth because your model will underfit sohow to find the best value, experiment because overfitting and underfitting are very subjective to a dataset, there is no one value fits all solution. So what I usually do is, let the model decide the max_depth first and then by comparing my train and test scores I look for overfitting or underfitting and depending on the degree I decrease or increase the max_depth.

min_samples_split: int, float, optional (default=2)

> *The minimum number of samples required to split an internal node:*

- *If int, then consider min_samples_split as the minimum number.*
- *If float, then min_samples_split is a fraction and ceil(min_samples_split * n_samples) are the minimum number of samples for each split.*

min_samples_split and min_samples_leaf, if you read their definitions it sounds like one would imply the other, but what you need to note is a leaf is an external node and the min_samples_split talks about an internal node and by definition an internal node can have further split whereas a leaf node by definition is a node without any children.

Say you specify a min_samples_split and the resulting split results in a leaf with 1 sample and you have specified min_samples_leaf as 2, then your min_samples_split will not be allowed. In other words, min_samples_leaf is always guaranteed no matter the min_samples_split value.

According to the paper, An empirical study on hyperparameter tuning of decision trees [5] the ideal min_samples_split values tend to be between 1 to 40 for the CART algorithm which is the algorithm implemented in scikit-learn. min_samples_split is used to control over-fitting. Higher values prevent a model from learning relations which might be highly specific to the particular sample selected for a tree. Too high values can also lead to under-fitting hence depending on the level of underfitting or overfitting, you can tune the values for min_samples_split.

min_samples_leaf: int, float, optional (default=1)

> *The minimum number of samples required to be at a leaf node. A split point at any depth will only be considered if it leaves at least min_samples_leaf training samples in each of the left and right branches. This may have the effect of smoothing the model, especially in regression.*

- *If int, then consider min_samples_leaf as the minimum number.*
- *If float, then min_samples_leaf is a fraction and ceil(min_samples_leaf * n_samples) are the minimum number of samples for each node.*

Similar to min_samples_split, min_samples_leaf is also used to control over-fitting by defining that each leaf has more than one element. Thus ensuring that the tree cannot overfit the training dataset by creating a bunch of small branches exclusively for one sample each. In reality, what this is actually doing is simply just telling the tree that each leaf doesn't have to have an impurity of 0, we will look into impurity further in min_impurity_decrease.

The paper, An empirical study on hyperparameter tuning of decision trees [5] also states that the ideal min_samples_leaf values tend to be between 1 to 20 for the CART algorithm. This paper also indicates that min_samples_split and min_samples_leaf are the most responsible for the performance of the final trees from their relative importance analysis [5].

According to scikit-learn, we can use min_samples_split or min_samples_leaf to ensure that multiple samples inform every decision in the tree, by controlling which splits will be considered. They also say a very small number will usually mean the tree will overfit, whereas a large number will prevent the tree from learning the data and this should make sense. I think one exception to this is when you have an imbalanced class problem because then the regions in which the minority class will be in majority will be very small so you should go with a lower value.

min_weight_fraction_leaf: float, optional (default=0.)

> *The minimum weighted fraction of the sum total of weights (of all the input samples) required to be at a leaf node. Samples have equal weight when sample_weight is not provided.*

min_weight_fraction_leaf is the fraction of the input samples required to be at a leaf node where weights are determined by sample_weight, this is a way to deal with class imbalance. Class balancing can be done by sampling an equal number of samples from each class, or preferably by normalizing the sum of the sample weights for each class to the same value. Also note that min_weight_fraction_leaf will then be less biased toward dominant classes than criteria that are not aware of the sample weights, like min_samples_leaf.

If the samples are weighted, it will be easier to optimize the tree structure using weight-based pre-pruning criterion such as min_weight_fraction_leaf, which ensure that leaf nodes contain at least a fraction of the overall sum of the sample weights.

max_features: int, float, string or None, optional (default=None)

> *The number of features to consider when looking for the best split:*

- *If int, then consider max_features features at each split.*
- *If float, then max_features is a fraction and int(max_features * n_features) features are considered at each split.*
- *If "auto", then max_features=sqrt(n_features).*
- *If "sqrt", then max_features=sqrt(n_features).*
- *If "log2", then max_features=log2(n_features).*
- *If None, then max_features=n_features.*

Note: the search for a split does not stop until at least one valid partition of the node samples is found, even if it requires to effectively inspect more than max_features features.

Every time there is a split, your algorithm looks at a number of features and takes the one with the optimal metric using gini impurity or entropy, and creates two branches according to that feature. It is computationally heavy to look at all the features every single time, so you can just check some of them using the various max_features options. Another use of max_features is to limit overfitting, by choosing a reduced number of features we can increase the stability of the tree and reduce variance and over-fitting.

As for among the options which one to pick, it will depend on the number of features you have, the computational intensity you want to reduce or the amount of overfitting you have, so if you have a high computational cost or you have a lot of overfitting, you can try with "log2" and depending on what that produces, you can either bring it slightly up using sqrt or take it down further using a custom float value.

random_state: int, RandomState instance or None, optional (default=None)

> *If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by np.random.*

Haha the notorious random_state, most novices ask me this, why 1, why 0 or why 42? 42 because that is the meaning of life, duh.

random_state is not really a hyperparameter to tune, or should you 😋. Let me start with when and why you should set a random_state. The most straightforward answer is so you can get consistent results, well somewhat because remember splitter, it will introduce some randomness to your results so if you rerun your decision tree, your results will be different, but it should not be too different.

That brings to my next point, I have seen new students play around with random_state values and their accuracy changes, it can happen because the decision tree algorithm is based on the greedy algorithm [6] where it is repeated a number of times using a random selection of features (splitter) and this random selection is affected by the pseudo-random number generator [7] which takes in the random_state value as a seed value, so by changing the random_state you might randomly pick good features, but what you need to realize is, random_state is not a hyperparameter, the changing of the accuracy of your model with the random_state just means there is something wrong with your model. It is a good hint that there are many local minima in your data and the decision tree is not dealing with it very well so I would rather have you set a random_state and tune your other parameters so that you don't get stuck in local minima than play around with the random_state.

min_impurity_decrease: float, optional (default=0.)

> *A node will be split if this split induces a decrease of the impurity greater than or equal to this value.*
>
> *The weighted impurity decrease equation is the following:*

```
N_t / N * (impurity - N_t_R / N_t * right_impurity
                    - N_t_L / N_t * left_impurity)
```

> *where N is the total number of samples, N_t is the number of samples at the current node, N_t_L is the number of samples in the left child, and N_t_R is the number of samples in the right child.*
>
> *N, N_t, N_t_R and N_t_L all refer to the weighted sum, if sample_weight is passed.*

min_impurity_decrease helps us control how deep our tree grows based on the impurity. But, what is this impurity and how does this affect our decision tree? Remember in the criterion section we quickly looked at Gini Index and Entropy, well, these are a measure of impurity. The impurity measure defines how well a number of classes are separated. In general, the impurity measure should be largest when data are split evenly for attribute values and should be zero when all data belong to the same class. A more detailed explanation will require us to go into information theory further which is not the scope of this article, so I will try to explain how changing the impurity values affect your model and how to know when to change these values.

The best way to tune this is to plot the decision tree and look into the gini index. Interpreting a decision tree should be fairly easy if you have the domain knowledge on the dataset you are working with because a leaf node will have 0 gini index because it is pure, meaning all the samples belong to one class. Then you can look into the splits that lead to 0 gini index and see if it makes sense to classify your classes as such or whether you can reduce the depth thereby leading to a more generalizable tree, if so, you can increase the min_impurity_decrease to prevent further division because now, the node will not be further split if the impurity doesn't decrease by the amount you specified. Note that this will affect your whole tree, so, you have to experiment with the numbers but the above explanation should give you a starting point.

class_weight: dict, list of dicts, "balanced" or None, default=None

> *Weights associated with classes in the form {class_label: weight}. If not given, all classes are supposed to have weight one. For multi-output problems, a list of dicts can be provided in the same order as the columns of y.*

class_weight is used to provide a weight or bias for each output class. But what does this actually mean, see when the algorithm calculates the entropy or gini impurity to make the split at a node, the resulting child nodes are weighted by the class_weight giving the child samples weights based on the class proportion you specify.

This can be highly useful when you have an imbalanced dataset. Usually, you can just start with the distribution of your classes as the class weights and then depending on where your decision tree lean, you can try to increase or decrease the other class weights so that the algorithm penalizes samples of one class relative to the other. The simplest way is to specify “balanced” and then go on from there with custom weights.

Note that this isn’t like an undersampling or oversampling technique, the number of samples in a class doesn’t actually change, its the weight assigned to it that does, you can see this when you print the decision tree and the values in each node, and it will change to

```
weight * (the number of samples from a class in the node) / (size of
class)
```

presort: bool, optional (default=False)

> *Whether to presort the data to speed up the finding of best splits in fitting. For the default settings of a decision tree on large datasets, setting this to true may slow down the training process. When using either a smaller dataset or a restricted depth, this may speed up the training.*

This parameter is fairly straightforward, if you have a small dataset or if you will restrict the depth of the tree and after running your first iteration you have an unbalanced tree where most data points are sitting on only a small portion of leaf nodes, using presort will help you.

The way presort works is it will initially sort all the variables before learning and at each node evaluation use sorted vectors, and after one choose the best split, you will split the data points and also the sorted indexes, in order to send to the child nodes the subset of data and subsets of sorted indexes. Thus you can apply this idea in recursion. Note however that this works if the number of data instances is greater than the number of input features.

Summary

The Decision tree complexity has a crucial effect on its accuracy and it is explicitly controlled by the stopping criteria used and the pruning method employed. Usually, the tree complexity is measured by one of the following metrics: the total number of nodes, total number of leaves, tree depth and number of attributes used [8]. max_depth, min_samples_split, and min_samples_leaf are all stopping criteria whereas min_weight_fraction_leaf and min_impurity_decrease are pruning methods.

Even though all of this pretty much implements either the stopping or pruning method, it varies on the level applied to the model. If you have a hard stopping criterion your model might end up under fitting so if you change it to a loose stopping criteria then your model may overfit, that is why we have the pruning methods. We should have a loose stopping criterion and then use pruning to remove branches that contribute to overfitting. But note that pruning is a tradeoff between accuracy and generalizability, so your train scores might lower but the difference between train and test scores will also get lower.

I hope you have a better idea of these parameters and how they might interact with each other when you are tuning the hyperparameters. But if something is not clear, please let me know in the comments and I would be more than happy to explain further.

Feel free to add me on LinkedIn or follow me on Facebook.

Reference

[1] Hyperparameter tuning for machine learning models.

[2] Scikit-learn DecisionTreeClassifier

[3] Laura Elena Raileanu and Kilian Stoffel, "Theoretical comparison between the Gini Index and Information Gain criteria" Annals of Mathematics and Artificial Intelligence 41: 77–93, 2004.

[4] Scikit-Learn Splitter Implementation

[5] Rafael Gomes Mantovani, Tomáš Horváth, Ricardo Cerri, Sylvio Barbon Junior, Joaquin Vanschoren, André Carlos Ponce de Leon Ferreira de Carvalho, "An empirical study on hyperparameter tuning of decision trees"
arXiv:1812.02207

[6] Greedy algorithm

[7] Pseudorandom number generator

[8] Lior Rokach, Oded Maimon, "Chapter 9: Decision Tree"

Sign up for The Variable

By Towards Data Science

Every Thursday, the Variable delivers the very best of Towards Data Science: from hands-on tutorials and cutting-edge research to original features you don't want to miss. Take a look.

Get this newsletter

Emails will be sent to shipingli@berkeley.edu.
Not you?

Machine Learning Data Science

About Help Legal

Get the Medium app

Download on the App Store

GET IT ON Google Play