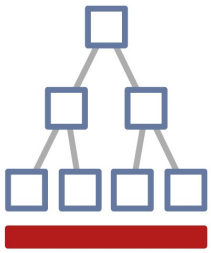
Decision Trees and Model Selection

What you'll do

Implement the CART algorithm to find the best split for a given data set and impurity function and build classification and regression trees
Choose the appropriate model that performs best for your problem and data using grid search and cross validation
Implement a machine learning setup from start to finish
Train and tune your classifier to maximize test accuracy



Course Description

The classification and regression tree (CART) is an effective algorithm for supervised machine learning. CART algorithms solve both regression and classification problems. During training, they recursively divide the space of the training data into small partitions - ideally so that each partition is particularly pure and contains primarily training points that share similar labels. During testing, CART trees identify which partition a test point falls into and returns the most common label of that partition (or its average in the case of regression). The partitions are stored in a compact tree data structure that requires very little storage and can be traversed extremely quickly during testing. Understanding the CART algorithm provides the foundation for more advanced machine learning methods such as Random Forests or Gradient Boosted Trees (often referred to by its most popular implementation XGBoost), which cleverly combine ensembles of many CART trees and are covered in later modules.

This course will explain how the CART algorithm can be derived and implemented. Further, Professor Weinberger will introduce you to different model selection methods and you will investigate their characteristics. You will practice selecting the best model and implement your own regression tree algorithm for a real world regression task.

Kilian Weinberger
Associate Professor
Computing and Information Science, Cornell University

Kilian Weinberger is an Associate Professor in the Department of Computer Science at Cornell University. He received his Ph.D. from the University of

Pennsylvania in Machine Learning under the supervision of Lawrence Saul and his undergraduate degree in Mathematics and Computer Science from the University of Oxford.

During his career, he has won several best paper awards at ICML (2004), CVPR (2004, 2017), AISTATS (2005) and KDD (2014, runner-up award). In 2011, he was awarded the Outstanding AAAI Senior Program Chair Award and in 2012 he received an NSF CAREER award. He was elected co-Program Chair for ICML 2016 and for AAAI 2018. In 2016 he was the recipient of the Daniel M Lazar '29 Excellence in Teaching Award.

Professor Weinberger's research focuses on Machine Learning and its applications. In particular, he focuses on learning under resource constraints, metric learning, machine-learned web-search ranking, computer vision, and deep learning. Before joining Cornell University, he was an Associate Professor at Washington University in St. Louis and previously worked as a research scientist at Yahoo! Research.

Q&A

1. Live Labs: Building a Community of Learners
2. Live Labs Q&A

Module 1: Create Decision Trees

1. Module Introduction: Create Decision Trees
2. Watch: Build a Decision Tree
3. Activity: Explore Decision Trees
4. Watch: Measure a Partition's Purity
5. Read: Formalize Impurity Functions
6. Activity: Determine the Minimum Entropy Split
7. Watch: CART Algorithm
8. Read: CART ID3-Algorithm
9. Tool: CART Cheat Sheet
10. Watch: Minimize Impurity of Regression Trees
11. Read: Squared-loss Impurity
12. Watch: Linear Time Regression Trees
13. Implement Regression Trees
14. Module Wrap-up: Create Decision Trees

Module 2: Choose the Right Model

1. Module Introduction: Choose the Right Model
2. Watch: Fine Tune a CART Tree
3. Read: Hyperparameters
4. Hyperparameters
5. Read: Underfitting and Overfitting
6. Watch: Find the Best Hyperparamter Setting
7. Watch: Pruning
8. Watch: Set Multiple Hyperparameters
9. Watch: Regulate the Complexity of a Classifier
10. Pick the Best Depth for your Regression Tree
11. Module Wrap-up: Choose the Right Model

Module 3: Improve Your Models

1. Module Introduction: Improve Your Models
2. Code: Load Data in Python
3. Tool: Loading the Data
4. Code: Explore Data in Python
5. Watch: Ensembling
6. The Value of Ensembling
7. Data Science in the Wild
8. Module Wrap-up: Improve Your Models
9. Read: Thank You and Farewell

Course Resources

Glossary

Q&A

1. Live Labs: Building a Community of Learners
2. Live Labs Q&A

Back to Table of Contents

Live Labs: Building a Community of Learners

Building a Community of Learners

This course will include two live (synchronous) video sessions called Live Labs. Live Labs will be offered each week as an opportunity to connect with your peers and instructor to address questions about the coursework and dive deeper into the material. Each hour-long session will be scheduled in advance. You are highly encouraged to take advantage of this chance to connect with your community of learners to build relationships and make this course experience even more rewarding.

This short Q&A section of the course contains a discussion called **Live Labs Q&A** where you can post questions and interact with your peers at any time as you work through this course. Common questions and important topics brought up in this discussion may be addressed in the Live Labs sessions.

In order to review the session schedule and join a session, you'll need to navigate to the **Live Labs** link under the **Course Shortcuts** section in the navigation menu on the left,

---

Back to Q&A

Back to Table of Contents

Live Labs Q&A

The Live Labs Q&A discussion is a place for you to post questions and interact with your peers throughout your work in this course. Common questions and important topics brought up in this discussion will likely be addressed in a Live Labs session.

Instructions:

Read through existing posts to find out what other students are discussing. Upvote posts that you find interesting or contain questions that you also have by "liking" it with the thumbs-up button at the bottom of the post.
If you have something to contribute to an existing discussion, reply to the thread and get involved in the conversation.
If you have new ideas, questions, or issues, post them in this Live Labs discussion board. If you have potential answers or suggestions that may resolve a question thread, feel free to share your understanding in a reply. This will help guide the instructor's feedback and discussion during their live session.

The questions that are the most upvoted or replied to that haven't been resolved will likely be addressed at the scheduled time by the instructor.

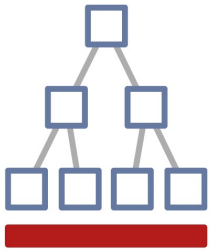
Back to Q&A

Back to Table of Contents

Module 1: Create Decision Trees

Back to Table of Contents

Module Introduction: Create Decision Trees



Classification and regression tree (CART) algorithms recursively split the data into partitions. You can keep track of these partitions in a tree structure. During testing, a test point traverses the tree until it falls into a leaf. Each leaf is associated with one of the data partitions, and you assign the test point the most common label within that partition (or the average label in the case of regression.) Professor Weinberger explains the conceptual and mathematical underpinnings of the CART algorithm. You will investigate how to partition the data into a tree that minimizes training error. You will also explore techniques to avoid overfitting and underfitting to the training data while maximizing your algorithms accuracy. In this module, you will have the opportunity to create your own regression tree.

Back to Module 1: Create Decision Trees

Back to Table of Contents

Watch: Build a Decision Tree

The k-Nearest Neighbor classifier is the simplest algorithm used in machine learning. While it is simple and can be very useful, k-NN takes much longer to run with larger data sets. Watch as Professor Weinberger explains how you can split your data set with a tree to improve the speed of computation. He also explains how finding the best tree for a given problem is a challenge, which we tackle later in the module with the CART algorithm.

Video Transcript

One of the first and simplest machine learning algorithm is the k-nearest neighbor classifier. The k-nearest neighbor classifier, you take a data set and during test time, you insert that test point into the data set. You find it's k-nearest neighbors, and you have a majority vote of the labels of those k-nearest neighbors. That's the label you assign. The one nearest neighbor classifier you simply just find the nearest neighbor, the most similar training point to the test point and you assign the label of that most similar training point. The k-nearest neighbor algorithm is great because it's really simple. You store your data during training and during testing you just do the nearest neighbor lookout. But it has a disadvantage. The disadvantage is that during test time you have to compute the distance to every single training point. That means it gets slower and slower the more training data you have. Imagine you have a billion data points, then you will have to compute the distance to a billion data points every single time you make a prediction. That can get really slow. So, what we will do now is we will try to come up with some clever data structure to do something very similar to the k-nearest neighbor classifier that's much faster. So, imagine for example, we take k-nearest neighbor classifier, but one thing I do is, I build a little wall, a little partition, and I divide the data set in half before I get the test point. Now I insert my test point, and I'm either in the right half [or] the left half. So, for example let's say I'm in the right half. Now what I do is assume because my test point is in the right half, probably my nearest neighbor will also be in the right half. So, I'm just searching the right half for my nearest neighbor and that's the label that I'm predicting. Tada! What I just did, is we just sped up the nearest neighbor algorithm by a factor two. Right? Because we only have to compute half as many distances. There's no reason to stop just after one partitioning. We could partition again and take the right half and partition it, for example, up and down, or further right—further left, et cetera, and we can take the left half and partition it again. So, let's say we partition this data set once again so we have four partitions. Now the algorithm is four times faster. We can do even better than that. That is, if you take into account what the labels of a certain partition are. If you do the partitioning in a smart way, then maybe we can partition until each single partition only

has one single label in it. So, in our case actually the top right partitioning only has red points. The bottom left partitioning only has blue points. So, if we enter our test point into their partition, we know that the nearest neighbor must be red because every single point in the partition is red. We don't even have to compute a single distance at all. We can immediately look up which partition we are in, then we turn the label of that partition. Essentially we are labeling the partitions beforehand during training. So, how do we know which partition a test point falls into? That again is actually quite simple. Let me partition the data set in half. What we're really doing is we're building a tree. So, initially we have a data set, S, and then we take the set and divide it in two, in the left set and the right set. Right? According to some splitting criteria, for example, if the feature value is greater than some threshold, then we take each one of these partitions and we partition them again. So, what we have is a tree structure and during test time all we need to do, is we take our test point— it initially isn't the original set—and then we descend down the tree looking, which partition do you fall into until you hit a leaf? And that leaf now has a label, either it is positive or negative—or red or blue. That's the label we return. The beautiful thing is during test time all we need to do is do this descending down the tree. That's all there is to it. Ideally, you would like to build the smallest possible tree such that every single leaf only contains pure labels. That means that every single data point inside that leaf has the same label. If you do that, then you basically descend and return exactly that label. Turns out, unfortunately, finding the exact smallest possible tree is a very hard problem. It's called NP-Hard. NP-Hard means you have to try out every possible tree there is to find the best one. But turns out there's a very good approximation. It's a greedy algorithm and it's called the CART algorithm. CART stands for Classification And Regression Tree. That's exactly what we will cover.

Back to Module 1: Create Decision Trees

Back to Table of Contents

Activity: Explore Decision Trees

To further explore how decisions trees work, you will complete two exercises that require you to review and make decisions based on some data. Our example data consists of "good" and "bad" individuals and some basic characteristics that describe each individual. Using the data, you will answer a series of questions about decision trees.

Part One: Training a Decision Tree

Take a look at the table below, which contains training data consisting of individuals (inputs), characteristics of those individuals (features) and whether that individual is "good" or "bad" (labels). Answer the questions that follow based on the data.

Training Data:

	Mask	Cape	Tie	Ears	Smokes	Height	Class
Batman	Y	Y	N	Y	N	180	Good
Robin	Y	Y	N	N	N	176	Good
Alfred	N	N	Y	N	N	185	Good
Penguin	N	N	Y	N	Y	140	Bad
Catwoman	Y	N	N	Y	N	170	Bad
Joker	N	N	N	N	N	179	Bad

Part Two: Using a Decision Tree

Take a look at the table below, which contains a few test data points consisting of individuals (inputs), characteristics of those individuals (features) but without labels. Based on our training data, we've provided a simple decision tree below that accurately classifies the training data. Use the data and tree to answer the following questions.

Testing Data:

	Mask	Cape	Tie	Ears	Smokes	Height	Class
Batgirl	Y	Y	N	Y	N	165	?
Riddler	Y	N	N	N	N	182	?

Fred	N	N	Y	Y	Y	181	?

Back to Module 1: Create Decision Trees

Back to Table of Contents

Watch: Measure a Partition's Purity

When building a tree, you want each partition to become “purer” (i.e., containing only data from a single class). If your partitions are pure, you can easily and confidently assign labels to new data points that lie within a partition. You use an impurity metric to measure a partition’s purity compared to other partitions. In this video, Professor Weinberger introduces two common impurity measures, the Gini Impurity and the Entropy Impurity. (Note: Professor Weinberger mistakenly refers to Gini Impurity as "Gini Index" at 3:56.)

Video Transcript

So when we build a CART tree, what we are essentially doing is, we take the data set and we repeatedly partition it in two sub-partitions. We would like to do this in a way such that the sub-partitions become purer and purer. Pure in a sense that they contain only points of one particular label. Ideally, when we actually have the leaves at the end, every single leaf only contains points of one exact label. There could be multiple leaves that have the same label but there should never be two labels in the same leaf. To guide us find these partitions, we need some kind of metric that tells us if partitions are purer than other partitions. We call this the impurity metric or the impurity measure. So, a very common one is a very popular one is the Gini impurity. The Gini impurity takes a data set and defines impurity by the probability that if I will take two points out of that data set randomly and their labels don't match. So how do we formalize this? Well, let's say we have, for example, have two classes, blue points and red points. Right, and now we take a data set, I randomly grab a data point, what's the probability that it's blue? What is the probability of the number of blue points divided by the total number of points, that's the probability of blue. What's the probability of red? Just the number of red points divided by the total number of points. In general, we can define pc as the probability that I will take a point of class C as the number of points with class C divided by the total number of points in that set. The Gini impurity then just sums over all these classes and multiplies the two terms pc times one minus pc. Here, pc is the probability that the first point has class C and one minus pc is the probably the

second point does not have class C. Therefore, that the two points actually have different labels. We can plot this function. So, in the case of two classes, we just have pc and one minus pc, so, let's say pc is the probability of blue and one minus pc is the probability of red. If you now plot this as a function of p blue then you can see that if p blue is zero, the Gini impurity is very low and if it's one, it's also very low. These are the two cases of perfect purity. In the case of one means every single point is blue, and the case of zero means that no point is blue, every single point is red. The Gini impurity has its highest point, its peak at exactly 0.5, that means that half the points are red, half the points are blue. That's the worst case, right that's when we're maximizing the probability that the two balls or two data points that we draw actually don't match. There are many other measures of impurity and a very common, very popular one is the entropy impurity. The entropy impurity is not quite as intuitive, but we can also plot it and you will see that it actually looks quite similar. Essentially what the entropy impurity is, is the sum of all the classes once again if pc or then you have times log of pc and you negate the whole thing. The entropy impurity reveals the history of decision trees which really come from information theory, and where originally designed is actually some kind of compression algorithm. What the entropy measures is essentially what's the probability if I had a sequence of data points that I draw out of this, out of a data set. Pc will be the probability that I draw a point with a certain class C, and log of pc would actually specifies the number of bits that I would need to communicate to you what the color or what the class of that particular data point is. The details are not very important, but essentially what we are doing with the entropy split is we are basically saying, you would like to have a data set that contains a lot of one class because then this data set is very compressible. Essentially that is something very similar to the Gini index and finds leaves that are predominately pure.

Back to Module 1: Create Decision Trees

Back to Table of Contents

Read: Formalize Impurity Functions

Gini impurity and entropy are two formal ways of capturing the label "disagreement" of a set.

Impurity metrics are higher for groups with mixed labels.

Impurity metrics are lowest when every point in a set has the same label.

As discussed in the video, you'll use one of two impurity functions — the entropy and Gini impurity — to quantify the heterogeneity of a subsection of data. Intuitively, the impurity is higher when a group of points has many different labels and lower when most points have the same label. Below, let's define the two functions more formally:

Impurity Functions

Data: $S = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}, y_i \in \{1, \ldots, c\}$, where c is the number of classes

Gini impurity

Let $S_k \subseteq S$ where $S_k = \{(\mathbf{x}, y) \in S : y = k\}$ (all inputs with label k)

$S = S_1 \cup \cdots \cup S_c$

The probability that a point picked uniformly at random from S has label k is therefore:

$$p_k = \frac{|S_k|}{|S|} \leftarrow \text{fraction of inputs in } S \text{ with label } k$$

Gini impurity of a leaf:

Each split in the decision tree partitions the data into two distinct sets and each training point falls into one of the leaves. The Gini Impurity for a leaf with corresponding set is the probability that two points, both picked uniformly at random from set , have *different* labels. Intuitively we sum over all possible labels and compute the probability that the first point has that particular label (which has probability) and that the second point does not have that label ( ).

$$G(S) = \sum_{k=1}^{c} p_k(1 - p_k)$$

Gini impurity of a tree:

The Gini impurity of a tree is defined recursively from the root. At a leaf the Gini Impurity is computed just as stated before on the training set falling into that leaf. For all intermediate nodes the Gini index is the weighted sum of the Gini index of the children (weighted by the fraction of training points that fall into the left or right sub-tree).

$$G^T(S) = \frac{|S_L|}{|S|} G^T(S_L) + \frac{|S_R|}{|S|} G^T(S_R)$$

where:

- $(S = S_L \cup S_R)$
- $S_L \cap S_R = \varnothing$
- $S_L \leftarrow$ Data points falling into left subtree.
- $S_R \leftarrow$ Data points falling into right subtree.
- $\frac{|S_L|}{|S|} \leftarrow$ fraction of inputs in left subtree
- $\frac{|S_R|}{|S|} \leftarrow$ fraction of inputs in right subtree

Entropy over tree:

The entropy over a tree is defined recursively (similar to the Gini index over a tree).

$$H(S) = p^L H(S^L) + p^R H(S^R)$$

$$p^L = \frac{|S^L|}{|S|}, p^R = \frac{|S^R|}{|S|}$$

Back to Module 1: Create Decision Trees

Back to Table of Contents

Activity: Determine the Minimum Entropy Split

In this activity, you will calculate the entropy to determine how to build the most effective decision tree, given a specific scenario and data set.

Scenario Setup

Imagine you are building a decision tree to predict whether a real estate loan given to a person would result in a tax write-off or not.

Your entire data set consists of 30 instances:
16 belong to the write-off class
14 belong to the non-write-off class
The data points contain two features, “Balance” and "Residence."
"Balance" can take on two values: “< $50K” or “≧ $50K.”
“Residence” can take on two values: “OWN” or “RENT.”

Based on our data in the parent node, we can calculate the entropy as follows:

$$E(\text{Parent}) = -\frac{16}{30}\log_2\left(\frac{16}{30}\right) - \frac{14}{30}\log_2\left(\frac{14}{30}\right) \approx 0.99$$

Activity

Your task is to determine which feature, "Balance" or "Residence," provides the lowest entropy split. There are two scenarios below to review. Once you've answered the question, click "Show Solution" to see if your answer is correct.

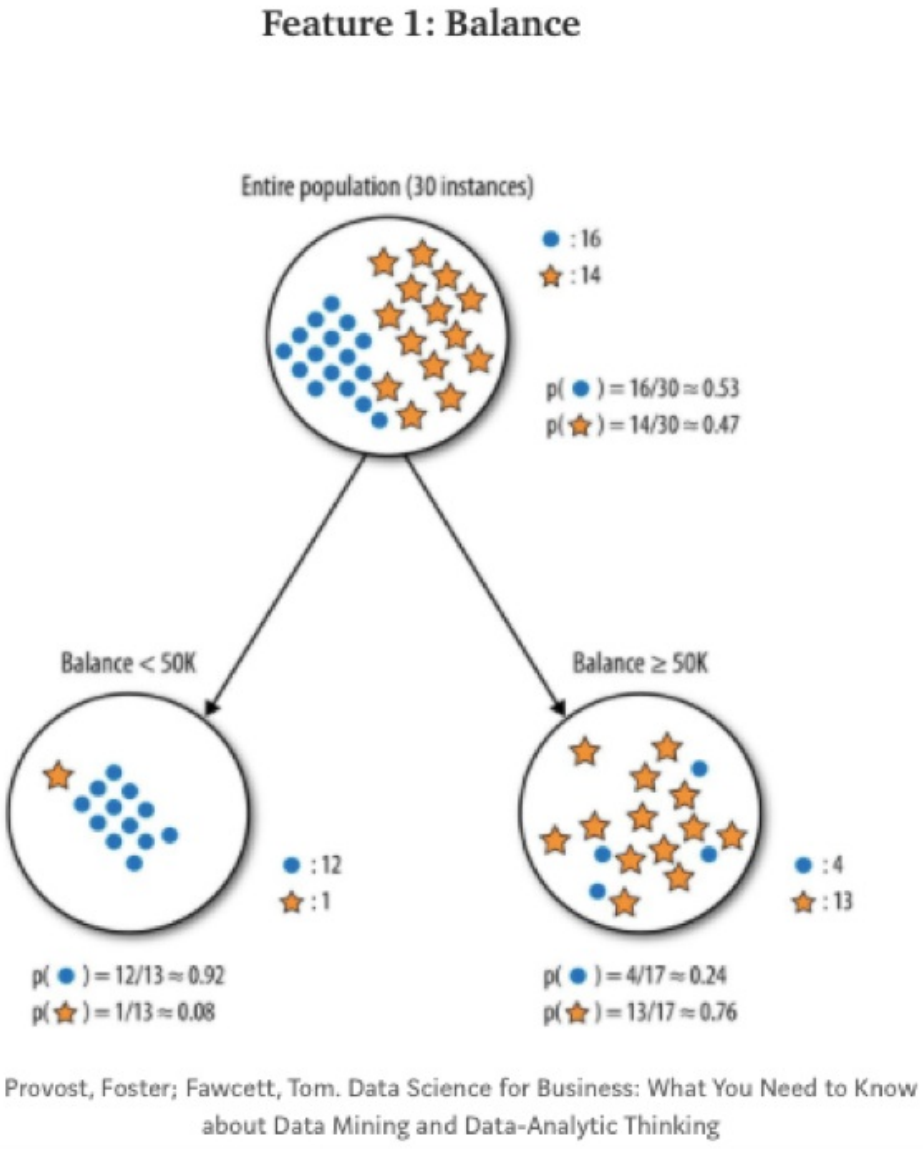
Scenario 1
Scenario 2

Split on Balance

Calculate the entropy for the parent node and see how much uncertainty exists by splitting on "Balance."

The blue circles are the data points with class write-off and the stars are the non-write-offs. Splitting the parent node on the "Balance" attribute gives us two child nodes: " $50K” or “ ≧ $50K". The "Balance $50K”node gets 13 of the total observations with 12/13 (0.92 probability) observations from the write-off class and only 1/13(0.08

probability) observations from the non-write of class. The right node gets 17 of the total observations with 13/17(0.76 probability) observations from the non-write-off class and 4/17 (0.24 probability) from the write-off class.



Provost, Foster; Fawcett, Tom. Data Science for Business: What You Need to Know about Data Mining and Data-Analytic Thinking

Question: What is the weighted average entropy for the nodes in this split?

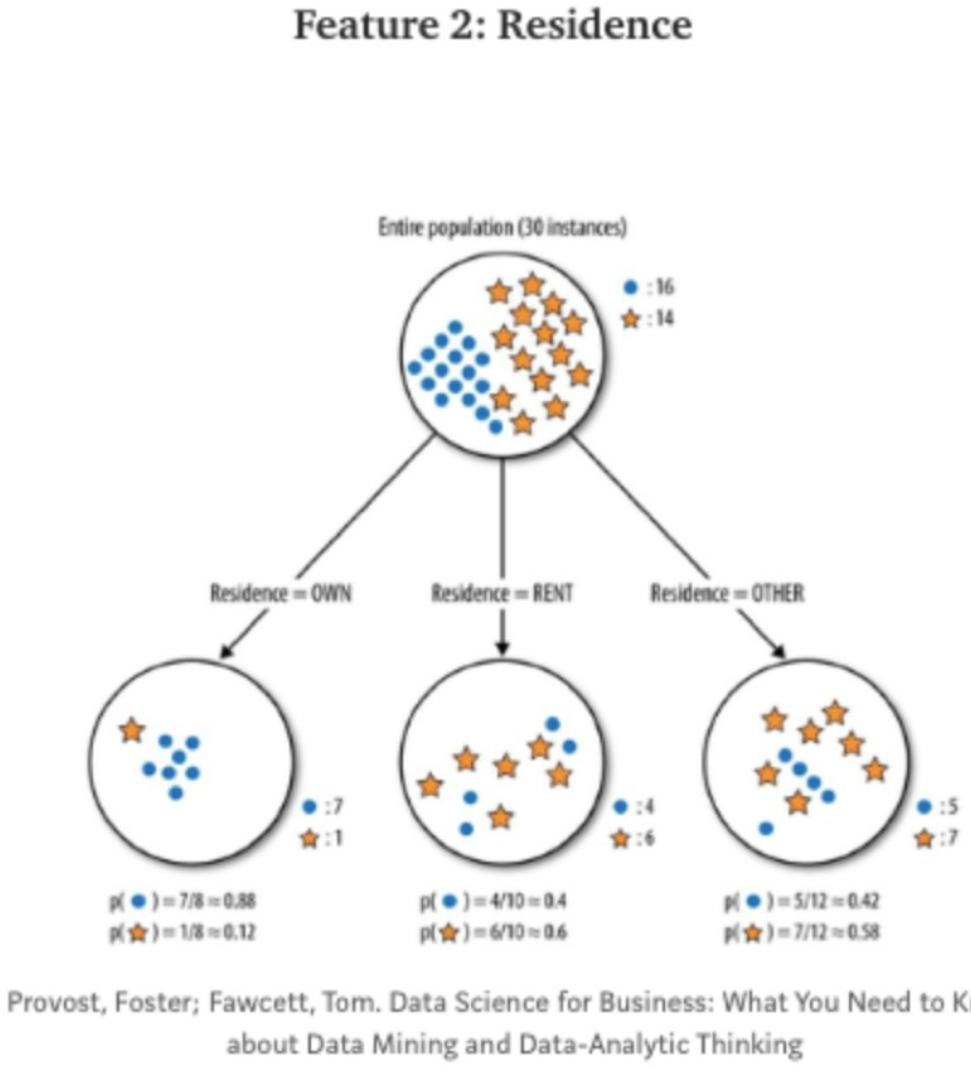
Solution

Before splitting, the entropy of the tree is 0.99. After splitting, the entropy of the tree is 0.62. So, by splitting on "Balance," the entropy is reduced by (0.99 - 0.62) = 0.37

Split on Residence

Calculate the entropy for the parent node and see how much uncertainty the tree can reduce by splitting on Balance

Splitting the tree on Residence gives you three child nodes. The leftmost child node gets 8 of the total observations with 7/8 (0.88 probability) observations from the write-off class and only 1/8 (0.12 probability) observations from the non-write-off class. The middle child nodes gets 10 of the total observations with 4/10 (0.4 probability) observations of the write-off class and 6/10(0.6 probability) observations from the non-write-off class. The right child node gets 12 of the total observations with 5/12 (0.42 probability) observations from the write-off class and 7/12 (0.58) observations from the non-write-off class.



Question: What is the weighted average entropy for the nodes in this split?

Before splitting, the entropy of the tree is 0.99. After splitting, the entropy of the tree is 0.62. So by splitting on Balance, the entropy is reduced by (0.99 - 0.62) = 0.37.

Back to Module 1: Create Decision Trees

Back to Table of Contents

Watch: CART Algorithm

Professor Weinberger discusses how the CART algorithm builds a decision tree by iterating through each value of each feature and identifying splits that result in the greatest entropy drop at each node.

Video Transcript

So now that we have a way to measure impurity in a data set, we'd like to use this metric to build a decision tree. Essentially, what we are doing is we repeatedly split the data sets and we find partitions such that the impurity improves. That means that the subpartitions now have a lower impurity than the original data set. That's exactly what the CART algorithm does. The CART algorithm is a way to find such a tree using an impurity metric. What it does, it tries out every possible partition in every single step and picks the best one. Let me walk you through an example. So, here we have five data points, they are blue and red. Well, the first thing to consider is how many different partitions are there. Well you could split after the first data point. Then you have the first data point go on one side, and all the other data points go on the other side, or you can split after the second data point, and the third data point, and so on. There is no point splitting after the fifth data point because now all the points would be on one side and not on the other. That's not really partitioning anymore. So, how many splits are there is exactly n minus 1 that we need to consider, and that's in each dimension. So, we have two dimensions. So, 2 times 4 is 8 different partitions. In general, if you have d times n minus 1, but d is number of dimensions and n is number of data points. All right, so now the CART algorithm essentially just goes through each one of these d times n minus 1 partitions and computes how good it is. So, let's look at the first one. So, we split off one point on one side and all the other points on the other side. Now, what we do is we compute the impurity on both of these sides. On one side, we're doing really well, but it only has one point. That point has only one label, so all the points on that side have the same label. This is a perfectly pure data set. The Gini impurity or the entropy impurity will be really, really low. In fact, it's exactly zero on that data set. On the other partition, however, it's not so good. Here we have two red points and two blue points. That's exactly 50-50. That's maximum impurity. That's a really, really bad data set. All right, that's not a good partition at all. So, is this now good or bad? One side is really, really good, the other one is really bad. So, what we do is we weigh the partitions by the probability of ending up in it. How do we approximate this probability? By saying, well, imagine you were a training point. What is the probability that you were in that partition? So, we just pick a training point at random and say what's the probability that you are right or that you are left. In this case, one partition would only have a probability of one over five, versus the other one

would have a probability of four over five. So, actually the split is not very good, right, because of the high probability you end up in a partition that has a very high impurity. More formally, what we're doing is for every single partition we compute the impurity on both sides, and then we weigh these impurities by the probability of ending up in that side which is exactly the number of points in that side divided by the total. We can compute this sum now for every single partition in every single dimension. Then you pick the one that has the lowest average; the weighted average impurity. That's the winner. That's the one we pick. We split the data set and now we recursively apply the algorithm on both of these subsets. When do we stop? The CART algorithm keeps splitting the data set until one of two things happens: Either we end up at a data set where every single point has the same label. Then we're already done, right, because this data set is perfectly pure, no reason to keep splitting. Or there's a second setting, if the data set is not perfectly pure, but the data points are all on top of each other. So, imagine you have two points, red and blue, but they have exactly the same features, right, you would like to split them, but you cannot because it's impossible. In that case, you also stop. So, once again either if it's pure or if all the data points have exactly the same features and you cannot split anymore, then you're done.

Back to Module 1: Create Decision Trees

Back to Table of Contents

Read: CART ID3-Algorithm

The CART-ID3 algorithm splits the data repeatedly to maximally improve the purity of its children.

If all possible splits have been exhausted or the node is already pure, a node will not be split further.

Because the same procedure is used on every node in the tree structure, the algorithm is recursive.

As discussed in the video, the CART algorithm builds a classification tree by iterating through all possible binary partitions of the training data based on a single feature threshold. It then picks the one partition that leads to the best weighted impurity of the resulting children. This procedure is repeated on each new data set resulting from each partition until a stopping criterion is met.

Here, we formalize this stopping criterion:

Base Cases:

The ID3 algorithm stops in exactly two cases:

1. If all data points in the data set share the same label we stop splitting and create a leaf with label y.

2. If all data points in the data set share the same features we create a leaf with the most common label y for classification and average label for regression.

Recursion:

If none of the stopping criteria are met, we try all features and all possible splits and select the split that minimizes the weighted impurity of the two children.

$$\frac{|S_L|}{|S|} I(S_L) + \frac{|S_R|}{|S|} I(S_R).$$

Here, the minimization is over the threshold t and the feature f, defining the two sub-partitions:

$$S_L = \{(x, y) \in S : x_f \leq t\}, S_R = \{(x, y) \in S : x_f > t\}$$

Formally we can state the ID3 algorithm as:

$$\text{ID3}(S)$$
$$: \begin{cases} \text{if } \exists \bar{y} \text{ s.t. } \forall (x, y) \in S, y = \bar{y} \Rightarrow \text{return leaf with label } \bar{y} \\ \text{if } \exists \bar{x} \text{ s.t. } \forall (x, y) \in S, x = \bar{x} \Rightarrow \text{return leaf with label mode}(y : (x, y) \in S) \text{ or mean (regression)} \\ \text{otherwise} \Rightarrow \text{return sub-trees} ID3(S_L), ID3(S_R) \text{minimizing} \frac{|S_L|}{|S|} I(S_L) + \frac{|S_R|}{|S|} I(S_R) \end{cases}$$

(Note the ∃ symbol means "there exists" and the ∀ symbol means "for all".)

Back to Module 1: Create Decision Trees

Back to Table of Contents

Tool: CART Cheat Sheet

Use this CART Cheat Sheet as a quick way to review the details of how the algorithm works.

This tool provides an overview of classification and regression trees for your reference. You'll find information about the applicability, underlying mathematical principles, assumptions, and other details of this algorithm.

Back to Module 1: Create Decision Trees

Back to Table of Contents

Watch: Minimize Impurity of Regression Trees

The CART algorithm can also be used to build trees for regression settings. Here, the tree approximates a real valued function with a locally constant function. A test point is predicted to have the average label of all training points that fall into the same leaf. Since all training data points in a regression problem will typically have different labels, you'll need to modify your approach to minimizing impurity. Watch as Professor Weinberger shows how this works with an example.

Video Transcript

So far we have assumed that the data points all have discrete labels like plus one or minus one, red or blue. They could also have multiple, it doesn't have to be binary. But CART tree, as it stands for classification and regression tree, naturally also extends to regression as the name suggests. For this we only have to make two little modifications. The first one is when we split the data set into partitions; in each partition, we simply predict the average label of all the training points that fall into that partition. The second thing is we need a different impurity metric. We can't use entropy or Gini because these look at the labels as how pure are the labels? But here all the labels will be different in a regression setting. Every single data point typically has a different value. So, we need to use something else. A very common way to measure impurity in a regression setting is simply to use the square loss. So, essentially what we're doing in each partition, we compute the average of all the labels in that partition. Then, we compute the impurity as what's the squared difference of every single label from that average on average. Let me go through an example. Here, we have some data points sampled from a function. These datapoints are just one-dimensional. They have one X-dimension, and they all have a label Y. If we split this function half, essentially what we are doing is we partition these points into two, and on each side, we now predict the average. That means we approximate this function by just two constant functions. If you split this again, we basically take these two constant functions and refine them more. Now, we have four different constant functions. In the limit, we have a little constant segment around every single training point. That's basically what we predict for any given test point.

Back to Module 1: Create Decision Trees

Back to Table of Contents

Read: Squared-loss Impurity

Regression trees use a modified impurity metric appropriate for continuous values.

This impurity quantifies the variance of the labels in a set.

The final tree predicts the mean label of all training points in each leaf.

As discussed in the video, we can tweak classification trees for continuous valued labels (regression). Because labels are no longer categorical, we redefine impurity such that it captures the "spread" of values in each node. Conveniently, we can capture this spread within a set by the variance of the labels within the set. The prediction made by a regression tree for a leaf with corresponding set S is simply the mean label y_S. With this mean label as the predictor, the variance impurity is identical to the squared loss:

$$I(S) = \frac{1}{|S|}\sum_{(x,y)\in S}(y-\bar{y}_S)^2 \leftarrow \text{Average squared difference from average label}$$

$$\text{where } \bar{y}_S = \frac{1}{|S|}\sum_{(x,y)\in S} y \leftarrow \text{Average label}$$

Finding the best split

Remember, you evaluate the quality of a split of a parent set Sp into two sets S_L and S_R by the weighted impurity of the two branches:

$$\frac{|S_L|}{|S_P|}I(S_L) + \frac{|S_R|}{|S_P|}I(S_R)$$

In the case of the squared loss, this becomes:

$$\frac{1}{|S_P|}\sum_{(x,y)\in S_L}(y-\bar{y}_{S_L})^2 + \frac{1}{|S_P|}\sum_{(x,y)\in S_R}(y-\bar{y}_{S_R})^2$$

During the CART algorithm we evaluate all possible splits at each parent node Sp. As |Sp| is constant for all of them, you can simply pick the split of a set Sp into S_L, S_R that minimizes:

$$\sum_{(x,y)\in S_L}(y-\bar{y}_{S_L})^2 + \sum_{(x,y)\in S_R}(y-\bar{y}_{S_R})^2$$

Back to Module 1: Create Decision Trees

Back to Table of Contents

Watch: Linear Time Regression Trees

Larger data sets take longer to compute and may result in more errors. Professor Weinberger explains how the Regression Tree algorithm can be made faster by introducing some clever bookkeeping tricks. The resulting algorithm scales (almost) linearly with the data set size, that means if the training data set doubles ,it requires only twice as much time to find the best split (rather than 4x in the naive implementation).

Video Transcript

When we build a CART tree, we essentially try every single split in every single dimension. For each split, we compute the impurity on the right side and on the left side, and we take a weighted average of those two impurity scores. That's the quality of that split. We try to find the best split. That's simple enough but it can be costly if the computation of the impurity scores is expensive. For example, for the regression loss, so we have squared loss regression, the impurities loss actually is the sum over all the labels on one side and we compute the average, sorry, the sum of the squared differences between the label and the average label. To compute the average label, you have to compute the sum of all the labels divided by the total and then we again have to sum over all the data points and compute the square differences. That's order and computations. So, every single split, we have to go over every single training point to compute the distance or the distance of the label and the average label and square it. So, we have d times n minus one splits, and for each one of these splits, you have to do n computations. So that's an n squared algorithm. That means if our data set gets twice as large, our algorithm takes four times longer. That's not good. We would like to run our classifiers over very large data sets, alright? So, ideally, you would like to have is some kind of close to linear algorithm. That means if you have twice as much data, the algorithm only takes twice as long. One way around this is to not just compute the impurity naively from scratch every single time, but to make some assumptions about in which order we examine the splits. Typically, we also don't just examine the splits arbitrarily, usually we go through them in order. For example, let's look at this data set here. Typically, let's say you want to look at the best split in the first dimension, what we do is we sort the features, then we just go through them in order. First thing we do is, we look at the first data point and say, what if we split that point off to one side or the other points to the other side? The second split we examine is, now we incorporate the second point and say, take two points that we split off to one side, all the other points go off to the other side. The next split is now three points split off to one side and so on. So, what we're doing here, is every time we examine a split, we essentially have the same setting as the previous split except for one data

point, which is now no longer on one side but has moved over to the other side, essentially. Also the point actually hasn't moved. The assignment has moved from one side to the other. What we would like to do is just keep track of this and instead of computing the impurity from scratch, what we do is, we take the previous impurity on the right side and the left side and now we take into account that, okay, we moved the boundary a little bit, so one point is no longer on the right side, it's now on the left side or the other way around, right? So, all we need to do is compute a little correction for both sides, a correction that only entails, only incorporates one data point that now has changed this assignment. In terms of the square loss, this actually is not very hard to do. The square loss is the sum of all the data points in that side and we compute the difference between the label and the average label and we square the whole thing. It turns out we can get away with just keeping three statistics about each side; the right side and the left side. Let's call them q, s and n. N is the simplest. N just says, so for the right hand side, it's the number of data points on the right side. It's very easy to keep track of this and if you move a point away from the right side to the left side, then we just decrement n on the right side and we increase it on the left side. S is just the sum of the labels on the right side on the left side. Again, if you have a data point in the right side you want to move it to the left side, we just subtract it on the right, move it, add it to the left. Q is the sum of the squared labels. So, if you have it on the right side and we move something, we just subtract a squared label of this data point and add it to the other side. Turns out we can actually rewrite the square loss entirely in these three terms. The way we obtain this is, we just take the square loss, we expand the square. So, now we get a sum over all the y i squared minus two times y bar times the sum over y i and then plus sum over y bar squared. One thing we can observe now is that y bar, actually, if we express that in terms of our new variables, y bar is just s over n. So, we can substitute an s over n every time you see y bar. So, y bar is gone. It's very simple to compute y bar. So, now everything is s over n or n to y i. If you look at the very first term, sum over y i squared, well that's just q, right? So we just have q there, and the second term is minus two times s over n times the sum over y i, that's exactly s right? So, in the middle we get minus two times s squared over n. On the right hand side, the last term is, here we just have a sum over s squared over n squared but the sum goes over n points. So, the whole thing is n times. There's actually no reference to i anywhere in this. So, we actually just take n times inside the sum. We just sum it up n times. So, what we get is s squared over n. So, if you now take the middle and the right term and combine them, we just end up with q minus s squared over n. Q minus s squared over n is the entire impurity on one side. So, all we need to do when we examine a new split, we update q, s, and n—super cheap—and then we compute the impurity again on the right side and on the left side, then we look at the next split, update q, s, and n and update the impurity on the right and the left side. This way, we turned a slow algorithm in a super fast algorithm.

Back to Module 1: Create Decision Trees

Back to Table of Contents

Implement Regression Trees

Back to Module 1: Create Decision Trees

Back to Table of Contents

Module Wrap-up: Create Decision Trees

In this module, you saw how dividing data repeatedly into partitions predicts labels in larger data sets more quickly than using the k-Nearest Neighbor approach. Professor Weinberger explained how applying the impurity metric to the partitioned data allows you to use the CART algorithm to build a tree with leaves containing data of increasingly pure labels. This allowed you to carry out the first code project of the course; implementing your own regression tree.

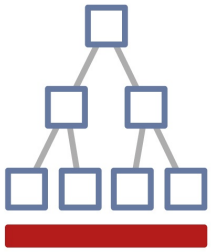
Back to Module 1: Create Decision Trees

Back to Table of Contents

Module 2: Choose the Right Model

Back to Table of Contents

Module Introduction: Choose the Right Model



Choosing an appropriate model and using the correct hyperparameters are critical when it comes to solving classification or regression problems. In this module, Professor Weinberger explains how hyperparameters effect classification and regression trees. He will also explain the roles of grid search and cross validation in fine tuning your algorithm. You'll have an opportunity to "tune" a regression tree yourself to put these concepts into action.

Back to Module 2: Choose the Right Model

Back to Table of Contents

Watch: Fine Tune a CART Tree

Hyperparameters are the configurable aspects of your algorithm that can be adjusted to improve its performance. For example, k-Nearest Neighbors has the hyperparameter “k” that determines the size of the neighborhood. As a data scientist, you must choose the correct hyperparameter setting to make your algorithm most effective. In this video, you will see the effects of incorrect hyperparameters and how to find the most appropriate settings given your situation.

Video Transcript

One great aspect of CART trees is that essentially you can always build a tree that gets zero training error on any real world data set. The reason is if there's any two data points that have a different label we split between them. But sometimes maybe that's not exactly what you want. Let me give you the following example. Here's a data set of blue points and red points, and if we build a tree it will learn to split the blue points and the red point apart from each other. But eventually to get zero training error what will it do? It will carve out a little region, a blue region within this red area just to get this one blue point right. But look at that blue point. It looks a little like a red point. It's right there in the center. It may just be a mislabeled datapoint and that's exactly what we call overfitting. Overfitting is when a classifier learns things that are true and hold on the training data set, but don't really hold in general. Like in this case probably there's no blue region right in that area of the space. It's probably just an artifact of noise. So, typically when we train CART trees, the training error goes down monotonic as the tree gets deeper and deeper and eventually hits zero. But if you look at the testing error and that's what we really care about at the end of the day, the testing error goes down initially but then eventually suddenly goes up again, and that's exactly what we want to avoid. That's the region where we're looking too much at the training data set and learn things that don't really hold in general. So, because you want to minimize the testing error we have to be careful not to underfit. That's the first region. That's the region where our classifier is not complex enough and we can't even do well on the training data set. So, we definitely also do badly on the testing data set and we also don't want to overfit. Overfit is what we do well on the training data set, but now we do badly again on the testing data set because we started memorizing things essentially from the training data set that don't hold on the testing set. We would like to find the sweet spot right in the middle but we don't underfit and we don't overfit. To find the sweet spot we have hyperparameters. In the CART tree for example we can introduce a hyperparameter that says, this is the maximum number of nodes that we allow to have in the tree, or the maximum number of splits, or the maximum depth. All of these are valid and a data scientist has to find the right hyperparameter setting such that we

get as close as possible to the sweet spot.

Back to Module 2: Choose the Right Model

Back to Table of Contents

Read: Hyperparameters

If you have to specify a model parameter manually, then it is probably a model hyperparameter.

Hyperparameters are the settings of an algorithm that can be adjusted to optimize performance. Same idea as turning the knobs of a radio to get a clear signal.

What is a hyperparameter in a machine learning model?

A model hyperparameter is a configuration that is external to the model and whose value cannot be estimated from data. Hyperparameters are often:

used to adapt a model to a particular setting.
specified by the practitioner.
set using heuristics.
tuned for a given predictive modeling problem.

You cannot know the best value of a model hyperparameter for a given problem and data set. You may use common approaches, copy values used on other models, or search for the best value by trial and error. When a machine learning algorithm is tuned for a specific problem, essentially you are tuning the hyperparameters of the model to discover the model that result in the most accurate predictions.

Model hyperparameters are sometimes also referred to as model parameters, which can make things confusing. A good rule of thumb to overcome this confusion is as follows: “If you have to specify a model parameter manually, then it is probably a model hyperparameter. ”

Examples of Model Hyperparameters

Model hyperparameters are the properties that govern the entire training process.

Below are some hyperparameters we have encountered throughout the class:

Size of neighborhood in k-NN.
Depth of tree in decision trees.
Step size in gradient descent.

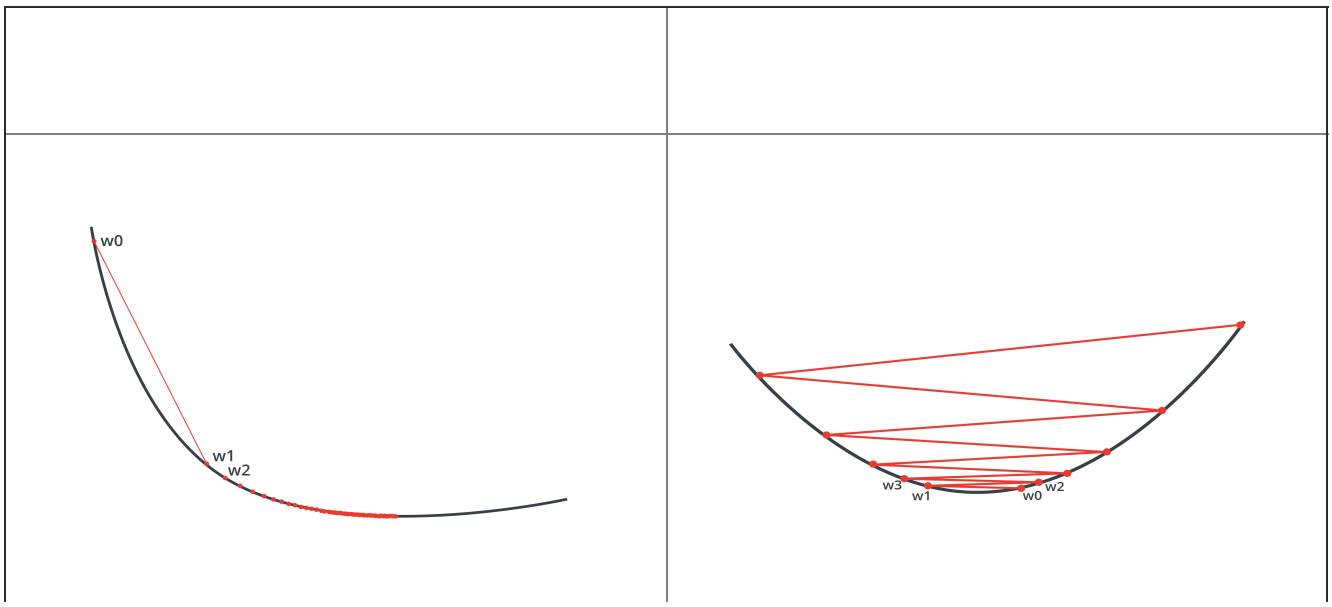
Let's take a closer look using the example of Gradient Descent:

Setting the **learning rate** in gradient descent can sometimes be more of an art than a science. Only if it is sufficiently small will gradient descent converge (see the first figure below). If it is too large the algorithm can easily *diverge* out of control (see the second figure below). A safe but often slow choice is to set a= t0/t, which guarantees that it will eventually become small enough to converge (for any initial value t0>0).

Why Hyperparameters Matter

The best way to think about hyperparameters is like the settings of an algorithm that can be adjusted to optimize performance, just as you might turn the knobs of a radio to get a clear signal. When creating a machine learning model, you'll be presented with design choices as to how to define your model architecture. Often, you won't immediately know what the optimal model architecture should be for a given model, and thus you'd like to be able to explore a range of possibilities. In true machine learning fashion, you'll ideally ask the machine to perform this exploration and select the optimal model architecture automatically.

Gradient Descent Convergence **Gradient Descent Divergence**



Back to Module 2: Choose the Right Model

Back to Table of Contents

Hyperparameters

Discussion topic:

Based on the type of data you intend to work with, describe a scenario where it might be easy to overfit to your data using a model like CART. Explain why there is a potential for overfitting and determine how you would prevent your model from overfitting.

Instructions:

You are required to participate meaningfully in all course discussions.
Limit your comments to 200 words.
You are strongly encouraged to post a response to at least 2 of your peers' posts.

To participate in this discussion:

Click **Reply** to post a comment or reply to another comment. Please consider that this is a professional forum; courtesy and professional language and tone are expected. Before posting, please review eCornell's policy regarding **plagiarism** (Links to an external site.)Links to an external site. (the presentation of someone else's work as your own without source credit).

Back to Module 2: Choose the Right Model

Back to Table of Contents

Read: Underfitting and Overfitting

An underfit model has not captured the predictive features present in the data and will perform poorly on new data.

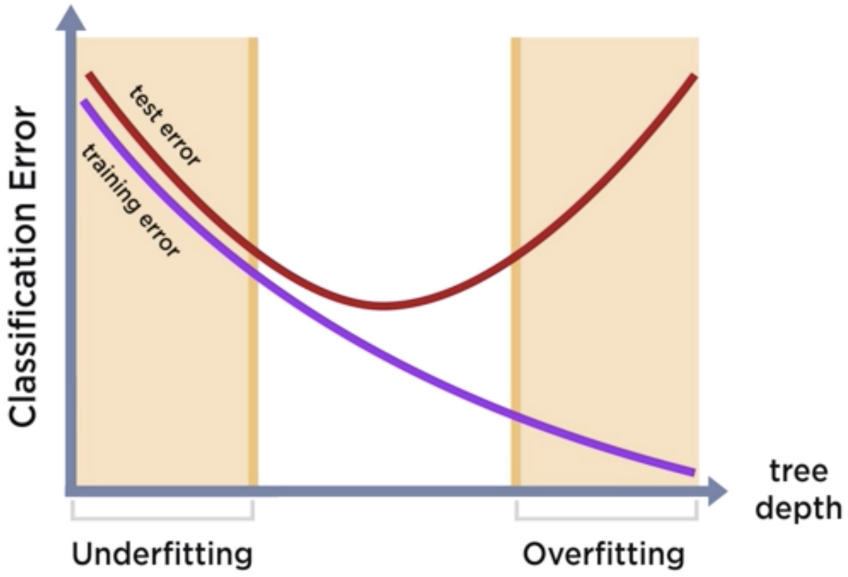
An overfit model has captured the idiosyncrasies of the training data and will not generalize to unseen data.

We can use plots of training and test error to assess the fit of our model.

There are two equally problematic cases which can arise when learning a classifier on a data set: underfitting and overfitting, each of which relate to the degree to which the data in the training set is extrapolated to apply to unknown data:

Underfitting: The classifier learned on the training set is not expressive enough to even account for the data provided. In this case, both the training error and the test error will be high, as the classifier does not account for relevant information present in the training set.

Overfitting: The classifier learned on the training set is too specific, and cannot be used to accurately infer anything about unseen data. Although training error continues to decrease over time, test error will begin to increase again as the classifier begins to make decisions based on patterns which exist only in the training set and not in the broader distribution.



Back to Module 2: Choose the Right Model

Back to Table of Contents

Watch: Find the Best Hyperparamter Setting

Cross validation is a method used to estimate the testing error of a machine learning model. Professor Weinberger explains how you can use this method to check and fine-tune your algorithm and hyperparameters.

Video Transcript

So how do we find the best hyperparameter setting to really hit the sweet spot between overfitting and underfitting? The problem is that we really want to minimize the test error, but we don't know what the test data is. Right? The test data is something you only see once the algorithm is deployed. We only have the training data. A common practice is to take the training data therefore and split it into two partitions: training and validation. Usually, it's maybe 80-20. So 80 percent is training, 20 percent is validation. When you train your classifier on this new training data set under different hyperparameter settings, and for each one of them you evaluate on the validation data set, you can plot this validation error and you pick the minimum. Now you know which hyperparameter setting gives you the lowest validation error and that's a good estimate of your test error. That can work really well if you have lot of data. If you have little data, it can be that the validation data is too small, right? Only 20 percent of the data. So, the validation error could fluctuate a lot based on which data points were chosen to be validation, which ones were chosen to be training. To counter that, a good trick is k-fold cross-validation. K-fold cross-validation, essentially what you do is you take your dataset and you split it into k buckets. Then you take k minus one of these buckets as training, and the kth as validation. So let's say k is five, then you basically split it into five buckets, each contains 20 percent of the data. You combine four of these buckets as training, and one as validation data set. You do this exactly five times. So essentially, every bucket becomes a validation set once. Each time, you try out a whole bunch of different hyperparameter settings, evaluated on the validation bucket that you left out. Now you have many such curves, you can average them and take the minimum of the average. That gives you a more stable estimate of your validation error. Typically what you do once you have this estimate of the best hyperparameter setting, you retrain your classifier one more time at the end on the full training data set, but you don't leave out anything and that's the model that you share.

Back to Module 2: Choose the Right Model

Back to Table of Contents

Watch: Pruning

Pruning is a technique used to speed up the CART algorithm by finding the right balance between complexity and simplicity in a decision tree. Watch as Professor Weinberger introduces different ways to prune.

Video Transcript

Decision tree classifiers are great because they are very simple to construct and very fast. But they have one disadvantage. And that is that it's hard to find the right trade off of complexity and simplicity. So essentially when you build a tree it's hard to tell, am I still underfitting or did I start overfitting? How big should my tree be? One simple way of finding the right balance is to do pruning. So pruning, what we do is we take our data set, split it in training and test—validation. And we build a full tree on the training data set until we get zero error. All the way, we don't worry about overfitting. And once we have this gigantic tree, we start pruning away nodes from that tree that have no or a negative effect on the error on the validation data set. And the way we do this is we do this bottom-up. So we start with the bottom of the tree and look at the different nodes and look at them and say, does this node actually help us on the validation set —not on the training set—on the validation set. And if it doesn't help us we just replace it by a leaf and say the tree ends here, and we do this bottom-up all the way and kind of examine all the different nodes all the way through the tree until we get the final tree that is usually a lot smaller. I can walk you through an example. So let's say we have this tree here that's trained on the training set and on the training data set. Of course each split helps us because otherwise we wouldn't have done it. But if you now look at the validation data set, we look at this tree bottom-up, we look at the first intermediate node that has some children. And we look at two cases. The one case is we just keep the tree as it is, the other case is we replace this particular intermediate node with a leaf node. Leaf nodes just predict the average label of all the training points that fall into it. We can just keep track of this for every single node, that's easy. So let's look at these two cases and see which one is better for the validation set. Let's say in this case actually turns out removing the children and just keeping the leaf node here is better for the validation set, so we just do that. And the tree now just got a little bit smaller. After this we move on to the next intermediate node on the same level and we examined that intermediate node. And again we go through it until we have examined all the intermediate nodes at that level, then we move one level up and look at all intermediate nodes here, et cetera. And we keep doing this until we've examined every single node in the tree and checked, does this node— the existence of this node really improve things on the validation set. There are other ways of pruning a tree. For example, some algorithms are a little bit more elaborate. They have a complexity trade

off and so on. But in practice what I've found is that just using this very simple pruning strategy where you just take a validation data set and say, I remove everything that doesn't help me on the validation set, typically works really well, is quite simple, and really easy to implement.

Back to Module 2: Choose the Right Model

Back to Table of Contents

Watch: Set Multiple Hyperparameters

Classification and regression trees have multiple hyperparameters: the number of nodes, the maximum depth, and the impurity function. There are various searches you can use to set the best hyperparameters, such as Grid Search, Telescopic Grid Search, and Random Search. In this video, Professor Weinberger will explain these various search methods.

Video Transcript

Sometimes machine learning algorithms have multiple hyperparameters that we need to set. In fact, CART trees do. One hyperparameter is the number of trees— number of nodes that we allow on the tree, the max depth. Another hyperparameter is the impurity function that we use. They could use the Gini impurity, or we could use the entropy impurity. When we have multiple hyperparameters to set, a common method to do so is—use grid search. In grid search, essentially what we do is we take the first hyperparameter, and think about several candidate values that we would like to explore. They make that the x-axis. We take second hyperparameter, make that the y-axis, and again we consider multiple hyperparameters that you want to explore. The combination of these values gives us a grid, and that's why it's called grid search. Essentially, what we do is—every single point on the grid, we evaluate. So we train a model with exactly this hyperparameter's configuration, we estimate the validation error, for example with k-fold cross-validation, and we store it somewhere. Once we're done we go through the entire grid and pick the setting with the lowest validation error, and that's the configuration that we use. Sometimes it helps to do telescopic grid search. That means, if you have a really good setting, and you zoom in one more time around that area, and do another grid search to really fine-tune your parameter setting. Also, if you have a lot of hyperparameters, then grid search can be a little bit slow. So, let's say you have 10 hyperparameters set. Well, then grid search takes a long time, because the number of points grows exponentially with the number of hyperparameters you have. In that case, often just simple random search works pretty well, where you essentially take the same space of hyperparameter settings, but you just pick random configurations, and evaluate them, and then pick the one that gives you the lowest validation error.

Back to Module 2: Choose the Right Model

Back to Table of Contents

Watch: Regulate the Complexity of a Classifier

Most hyperparameters regulate the complexity of the classifier. Professor Weinberger explains two paradigms for regulating this complexity trade-off, regularization and early stopping. When training CART models, we can use a variant of early stopping to prevent overfitting.

Video Transcript

What most hyperparameters in machine learning are really doing is, they are regulating the capacity or complexity of a classifier. You can envision them as a big knob that you can turn. If you turn it in one direction, you increase the complexity. If you turn it to the other direction, you decrease the complexity. Our goal is to find right the sweet spot. If it's too complex, then a classifier can essentially memorize the entire chain data set. For example, in decision tree that would be a decision tree with full depth that can just have one leaf for every single training data point. If it's too small, if a decision tree maybe only has one or two splits, it won't be able to capture the complexity of the data set, and we'll underfit. This means that the classifier is not complex enough. There's two different paradigms of how to regulate this complexity. One is regularization, the other one is early stopping. In regularization, what we do is we introduce a penalty that says—classifiers, if they get too complex are penalized. So, for example, one way of doing this in decision tree—the simplest way is to say, we each put a hard stop on it. Saying, if we have a maximum depth of maybe five, and we can't have a bigger tree than that. Even more fancy is, change the impurity function. Where we say, the impurity function is for example, entropy impurity, plus a term that says lambda times the number of nodes that we have in the tree. So, every time we split, we have to think about, does the reduction in impurity— is that lower than the increase in the penalty that we get for introducing additional nodes? In other learning paradigms, for example, if you do gradient descent of linear classifiers, if you train a logistic regression classifier, or an SVM, we have a very similar setting. If you do regularization, we minimize the loss of the classifier, plus a term that measures the complexity. Often, what people do is, for example, the L2 norm. So, you have lambda times the L2 norm. So, if your classifier is too complex, that means the w vector has a large L2 norm, then this term is very large. As you do gradient descent, you typically increase the norm. So, eventually basically, it doesn't become worth it anymore. So, regulating this lambda in these two cases, is a way of finding the sweet spot, and lambda here is an explicit hyperparameter that trades off complexity versus simplicity. Another way to regulate the complexity is not to introduce a regularization term, but instead do early stopping. Early stopping basically says if we learn a classifier, and our algorithm, this only works in these cases. If our algorithm is such that as we iterate

through our learning algorithm, our classifier becomes more and more complex, at some point we just stop and say now, this is enough complexity. The way we do this early stopping is by looking at the validation data set. So, in CART trees what you would do is you would say, I don't have a maximum depth, but what I am doing is I keep splitting my data set, I'm building the tree. But every time I split I peek at the validation set and say, how well does the current decision tree do? So, essentially what you could do is you could plot this graph, this is saying well the x-axis is the number of splits of the number of nodes you have, and you're plotting the error on the validation data set, and you keep learning your tree, but you keep peeking, and somebody will say, well wait a second. My validation error is going up again. That's when you stop, and usually you recover the point that was the lowest. So, if you're smart you kind of save, every single time you save your current setting so that you can look back and say okay that was actually the lowest point, so I just keep that one. In gradient descent, you can do exactly the same thing. So, every single time you do a gradient update of your weight parameters, you peek at the validation set and say, well how well am I doing right now? And once you think you've surpassed the minimum, you recover the minimum vectors by undoing steps, or by recovering something that you saved onto disk, and that's the classifier that you return. In practice, people use both. For some classifiers, you use regularization, and sometimes you use early stopping. Sometimes people even do both.

Back to Module 2: Choose the Right Model

Back to Table of Contents

Pick the Best Depth for your Regression Tree

Back to Module 2: Choose the Right Model

Back to Table of Contents

Module Wrap-up: Choose the Right Model

In this module, you explored hyperparameter setting and reviewed two paradigms for regulating complexity, regularization and early stopping. Remember that regulating the complexity of the classifier by using the correct hyperparameters avoids both overfitting and underfitting, improving the likelihood that your model performs well on new data. You were able to apply these practices with the next part of your project by implementing k-fold cross validation. You now have the tools to select and tune your models. This next module gives you the opportunity to put it all together from scratch using a live data set.

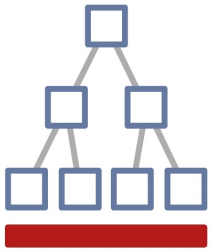
Back to Module 2: Choose the Right Model

Back to Table of Contents

Module 3: Improve Your Models

Back to Table of Contents

Module Introduction: Improve Your Models



Data scientists want their programs to perform optimally, leading to results with the lowest possible error on a data set. There are various methods you can use to make your code better. In this module, you will start by loading and exploring a data set. Professor Weinberger will then review the different techniques data scientists use to improve their models and you will practice using these methods to improve your own models.

Back to Module 3: Improve Your Models

Back to Table of Contents

Code: Load Data in Python

Load Data in Python

Back to Module 3: Improve Your Models

Back to Table of Contents

Tool: Loading the Data

Use the Loading Data guide as a quick way to review the details on loading data.

This tool provides an overview of loading various data types for your reference. You'll find information about different functions and libraries fit to your data type.

Back to Module 3: Improve Your Models

Back to Table of Contents

Code: Explore Data in Python

Explore Data in Python

Back to Module 3: Improve Your Models

Back to Table of Contents

Watch: Ensembling

Professor Weinberger explains different ways of combining machine learning classifiers. Combining classifiers reduces error and can make your model more adaptable, given your settings.

Video Transcript

A simple way to make machine learning classifiers generalize better, work better on the test data set, is to combine different kinds of learning paradigms. For example, you could use logistic regression, you could use k-nearest neighbors, you could use decision trees. Why not combine them into one uber classifier? One way of doing this is to, for example in the case of decision tree, learn a decision tree, but in the leaves don't just predict the most common label. Instead, take all the twin data points that fall into this particular leaf, and train a classifier in that leaf. Of course this only works if your tree doesn't go all the way to the bottom. So, let's say you only build a tree of maximum depth; three, for example. Now you do this trick that for each leaf you actually train a linear regressor. Essentially what you're doing if you have a data set and a regression problem, you split the data set into many different parts and then for each individual part, you learn a linear regressor. This way you can turn a linear classifier into a nonlinear classifier. The decision tree does the nonlinear part. It splits the data set into portions that are roughly approximately linear and then the linear regressor fits this part really well. This is very very effective and can reduce your error by quite a bit. When you do, when you build a decision tree, typically we also use the loss of the linear regressor as the impurity. Another way of combining classifiers is to use ensembling. Ensembling has the advantage that you wouldn't have to change any code. You can just use existing code. Let's say you have a data set and you don't know, should I use k-nearest neighbors, should they use logistic regression, should I use decision trees— what you do is you just run all three algorithms. And during test time, you evaluate each one of these three algorithms and you do a majority vote amongst those classifiers. The trick is that different kinds of classifiers make different kinds of mistakes. It's actually quite rare that they all get the same data point wrong. So, typically, when you have a test point of one classifier gets it wrong, well the other two probably get it right. This way you reduce the overall error rate of your classifier.

Back to Module 3: Improve Your Models

Back to Table of Contents

The Value of Ensembling

Discussion topic:

Discuss the intuition behind ensembling as a machine learning technique.
Based on what you've learned about different types of classifiers so far, what are some real-world applications of ensembling?

Instructions:

You are required to participate meaningfully in all course discussions.
Limit your comments to 200 words.
You are strongly encouraged to post a response to at least 2 of your peers' posts.

To participate in this discussion:

Click **Reply** to post a comment or reply to another comment. Please consider that this is a professional forum; courtesy and professional language and tone are expected. Before posting, please review eCornell's policy regarding **plagiarism** (the presentation of someone else's work as your own without source credit).

Back to Module 3: Improve Your Models

Back to Table of Contents

Data Science in the Wild

Back to Module 3: Improve Your Models

Back to Table of Contents

Module Wrap-up: Improve Your Models

In this module, you used live data to build your own model, from start to finish. You practiced loading and exploring data as well as combining classifiers to reduce error and complexity of your CART implementation. You now have all of the pieces necessary to select, implement, and improve your machine learning model.

Back to Module 3: Improve Your Models

Back to Table of Contents

Read: Thank You and Farewell

Kilian Weinberger
Associate Professor
Computing and Information Science

Cornell University

Congratulations on completing "Decision Trees and Model Selection."

I hope that you learned some crucial machine learning concepts. Decision trees are the "gateway" to many advanced machine learning algorithms, and model selection is they key to successful data science. I hope the material covered here has met your expectations and prepared you to better meet the needs of your organization.

From all of us at Cornell University and eCornell, thank you for participating in this course.

Sincerely,

Kilian Weinberger

Back to Module 3: Improve Your Models

Back to Table of Contents

Glossary

Classification and regression tree (CART)

A popular type of supervised learning algorithm, used for example in search engines, and relying on partitioning of data into nodes. Using an impurity metric, the algorithm builds a tree data structure where leaves (end points) correspond to regions in the data space that are dominated by one class label.

Early stopping

One of two paradigms (the other is regularization) for governing complexity in a classifier. Early stopping works with classifiers that increase model complexity during learning. We evaluate the classifier on the validation data set in regular intervals as the complexity of the model increases. We stop the algorithm at the point where the validation error is the lowest (by saving the model for each evaluation), right before the validation error starts to rise again.

Ensembling

A method of combining classifiers by running multiple algorithms. You can use the majority vote of the classifiers to evaluate the algorithms.

Entropy impurity

A popular impurity metric used in building a CART. The entropy of a set decreases as it is dominated by just a few labels and is minimized (zero) if all elements of a set share the same label. Entropy has its origins in information theory and corresponds to the number of bits required (in expectation) to encode the label of a point.

Gini impurity

A common impurity metric used in building a CART. Expresses impurity by the probability that the labels of two points in a partition do not match.

Greedy algorithm

A simple algorithm used in optimization problems. It makes the best choice at each step (which may however not be optimal globally) in finding the solution to an entire problem. The CART algorithm is a greedy algorithm.

Grid search

A common method for setting multiple hyperparameters. Given multiple hyperparameters and possible candidate settings, Grid Search trains a model with all possible configurations and evaluates them on a validation data set. We ultimately select the setting with the lowest error.

Hyperparameter

A parameter in the model that is not learned but set prior to learning. Hyper-parameters often trade-off complexity vs. simplicity of models. There are many heuristics to set hyper-parameters. A common one is to use grid-search or early-stopping.

K-fold cross validation

A method for taking a dataset and splitting it into K buckets. These buckets are then divided into K-1 training buckets and one validation bucket. This method allows each data point to be in a validation data set once.

Leaf

A node where a decision tree ends, that predicts the average or most common label from the training data in that leaf.

Model selection

The process of selecting an algorithm or hyper-parameter setting for a particular dataset / task.

NP-hard

An expression of hardness of a problem, non-determisitc polynomial-time hardness. In building decision trees, this means we would have to try out every possible tree to find the most compact tree. The CART algorithm allows us to approximate the smallest tree.

Overfitting

A cause of error in classification of data points that occurs when the classifier learns things that are true and hold for the training dataset but do not hold in general.

Pruning

A process for reducing the size of a tree by removing nodes that have no or a negative effect on the error for the validation dataset.

Regularization

One of two paradigms (the other is early stopping) for governing complexity in a classifier. A penalty is applied to classifiers if they become too complex, for example, with a hard stop or by changing the impurity function. We minimize the loss of the classifier plus a term that measures the complexity.

Simple random search

An alternative to grid search for choosing multiple hyperparameters, by evaluating random hyperparameter configurations, finding the validation error, and selecting the configuration with the lowest error.

Underfitting

A cause of error in classification that occurs when the classifier is not complex enough and is unable to obtain a low error even on the training dataset.