# **Machine Learning with Python: Decision Trees**

### **Making decisions with Python**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Fred] Decision trees are one of the most popular and easy to use approaches in supervised machine learning. So how does a decision tree get built? When should I use one? Can they get too large and complicated? If so, how do I make them more concise? Hi, I'm Fred Nwanganga, I'm a data scientist, teacher, and author with a passion for AI and machine learning. In this course, I introduce what a decision tree is. I explain the mechanics of how decision trees are pruned. I discuss when decision trees are most useful, and when they are not. Finally, I walk through the process of building, visualizing, and optimizing a classification tree, as well as a regression tree in Python, let's get started.

### **What you should know**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] Before we get started, let's go over some of the background knowledge that you should have in order to get the most out of this course. First, I assume that you have a fundamental understanding of what machine learning is, specifically supervised machine learning. If you don't, I recommend that you review my introductory LinkedIn Learning course, Machine Learning with Python: Foundations. Second, it will be helpful if you have some entry level knowledge coding in Python I will assume that you know how to import a Python package, refer to attributes of an object and call the methods of an object. Next, it would also be helpful if you have some familiarity with the pandas and scikit-learn packages. If you don't, no worries. I will explain exactly what I'm doing when we do use these packages in the course. Finally, I do assume that you know how to use the Jupyter Notebook interactive Python environment. Specifically, I assume that you know how to create a code cell as well as how to edit and run code within a code cell.

### **The tools you need**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Man] Before working through the exercises in this course, I want to make sure that you have the tools you need, to be successful. The first thing you want to make sure to do, is install or verify that you have the latest version, of Python 3. I'll be working in Python 3.8. So if you have the latest version of Python 3, you should be able to run my code. However, if you're still on Python 2, you may run into some difficulty. Second, my code will be written and executed, using Jupyter Notebooks. The Jupyter Notebook, is a great platform for combining live code, descriptive text about the code, visualizations, and equations in one simple-to-use interface. Luckily, everything you need, which includes the latest version of Python 3, and the Jupyter Notebooks environment, is available by installing Anaconda. Anaconda is a package manager, an environment manager, a Python distribution, and a collection of over 7,500 open-source packages. It is free and easy to install, and it offers free community support. You can install Anaconda, by going to the address provided here.

### **Using the exercise files**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] The exercise files you need for this course will be provided to you. This means that you can follow along with any of the code examples in the lessons. I recommend that you do so. The best way to become proficient in building decision tree models in Python is to practice doing it yourself. The exercise files are organized into folders that correspond with the chapters of the course. Within each folder are data files and two notebooks for each of the code lessons. Let's take a look at an example. As you can see, folder 02 has a data file called loan.CSV. It also has two notebooks for the chapter two lesson videos. The 02B notebook is the beginning notebook. This is the notebook you should code in when following along with the lesson videos. The 02E notebook is the ending notebook. It is a completed version of the beginning notebook for your reference. There are several ways to launch a notebook. One approach is to launch the Anaconda Navigator, click on Launch Jupyter Notebook, then navigate to the notebook you want and launch it. Let's do this together. Depending on how fast or how resourced your computer is, this may take a while, or it may happen very quickly. Now that we have Anaconda Navigator open, let's click on Launch Notebook. Then we go to the folder, find the exercise files, and now we can launch our notebook.

### **What is a decision tree?**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] A decision tree is a machine learning approach that uses an inverted tree-like structure to model the relationship between independent variables and a dependent variable. Decision trees are a supervised machine learning approach. This means that we can use a decision tree to predict future outcomes. Because of their versatility and ease of use, decision trees are one of the most widely used machine learning approaches. Just like a tree has a trunk which is connected to leaves by branches, a decision tree is a collection of decision nodes which are connected to other decision nodes or leaf nodes by branches. We can also think of a decision tree as a collection of questions, responses, and outcomes. To illustrate this point, let's assume we are thinking of getting a new job. For us, the choice of whether or not to accept a new job offer hinges on certain key considerations. Let's assume that the most important consideration is salary, specifically we only accept a job offer if the job pays more than $80,000 a year. In a decision tree, this first consideration or question is represented as a decision node. Since the answer to this question is the most indicative of whether or not we will accept an offer, it is the first decision node in our tree and is thus known as the root node. The possible responses to the question are represented by a branch for yes and a branch for no. And depending on the answer, we either accept a job or reject it. The outcome of each response, accept or reject, is represented by a leaf or terminal node. Now let's assume that commute time is also an important consideration as to whether we accept a job offer or not. Specifically, even if a job pays more than $80,000 a year, we may not be willing to take it if the commute is longer than an hour from where we live. In a decision tree, we model this by adding a second decision node to the tree. We now have two decision nodes to possibly consider before getting to the leaf nodes of accept or reject. Next, let's assume that paid time off is also very important to us. Specifically, we would accept an offer that pays less than $80,000 a year as long as it offered more than four weeks of paid leave. We model this question with a third decision node. We now have three decision nodes to possibly consider before getting to the leaf nodes of accept or reject. As more considerations come to mind, we can add them as subsequent decision nodes which would increase the size of our tree. However, if we decide that these three considerations are adequate, then this tree serves as a predictive model for whether or not we will accept a given job offer. This structure of a decision tree can be interpreted as a set of rules or guiding principles. For example, if we provide this model to a recruiter, they could easily follow the logic of this tree from the root node to the leaf node to figure out whether any given job offer would be acceptable to us or not. Now imagine for a moment what your job acceptance decision tree could look like. I'm sure it would have more decision nodes than this one. I previously mentioned that decision trees are one of the most widely used machine learning approaches. A major reason for this is the ease with which decision trees can be translated into simple and understandable if-then-else rules. This makes them well suited for situations where transparency is important for legal or compliance purposes and for situations where the decision logic needs to be shared with non-technical stakeholders. As a supervised machine learning approach, decision trees can be used to solve both classification problems and regression problems. If our dependent variable is a categorical or discreet value such as color, true or false, yes or no, then the type of tree we build is called a classification tree. However, if our dependent variable is a continuous value, such as age, income, temperature, then the type of tree we build is called a regression tree. In this course, we will use both types of trees.

### **How is a classification tree built?**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] Classification trees are built using a process known as recursive partitioning. The basic idea behind this process is to repeatedly split data into smaller subsets in such a way that maximizes the homogeneity or similarity of items within each subset. To illustrates how recursive partitioning helps us build a classification tree, let's imagine that we work for a small commercial bank and that we have historical data for 30 personal loans issued by our bank. Each loan record includes the annual income of the borrower, the amount that was borrowed, and the outcome of the loan, which is represented here by the default column. Note that the income and loan amount columns are what we call the independent variables or predictors while the default column is a dependent variable or class. Each of the 30 loans previously issued by our bank can be represented in terms of the dependent and independent variables this way using a scatter plot. From the plot, we can see that of the 30 loans in the dataset, 16 ended in default, the red triangles, and 14 were paid back in full, the green circles. Recall that the idea behind recursive partitioning is to repeatedly split data into smaller subsets in such a way that maximizes the similarity of items within each subset. So the first thing we need to do here is to figure out how best to split this data into two so that we have partitions or subsets that maximize the similarity or purity of outcomes. Using two axis parallel lines, we scan both axes to determine where to split the data. By visual inspection, we find that splitting on the loan amount of $40,000 gives us the best split. Based on the split, we get 14 loans of $40,000 or less to the left and 16 loans of more than $40,000 to the right. Splitting the data this way gives us the two partitions with the most homogeneity of loans in favor of one of the two outcomes. Any other axis parallel line we could have drawn would result in partitions with less purity or homogeneity. Notice that I use the terms homogeneity, similarity, and purity to represent the same idea. This initial split creates the logic for the root node of our classification tree, which is shown here. It simply asks the question, did a customer borrow $40,000 or less? To create the branches and the next set of nodes, we make some generalizations or simplifying assumptions. Of the loans which were more than $40,000, 10 resulted in default while six were paid back in full. In other words, 63% of the loans, or 10 out of 16 loans, in this partition resulted in default. Because default is the dominant outcome in this partition, we will assume or generalize that any future loans that are for more than $40,000 will also result in default. As you can see, some of the loans in the partition, the red circles, do not conform to our assumption. We refer to these as the misclassified examples in the training data. Our goal should be to have very few of these. The assumption we made for this partition determines the structure of the first branch and leaf node in our classification tree. Now let's take a look at the other partition. Of the loans which were $40,000 or less, eight out of 14, or 57% were paid back in full while six resulted in default. Because not default is the dominant outcome in this partition, we generalize that any future loans that are $40,000 or less will also be paid back in full. As expected, we also have some misclassified examples in this partition. These are the green triangles. The generalization we made for this partition determines the structure of the second branch and leaf node in our classification tree. We can stop the recursive partitioning process here or we can decide to keep trying to create purer partitions within the data. For instance, we know that within the left partition, we misclassified six of the 14 examples. To reduce the number of misclassified examples, we need to further partition the data. Using two axis parallel lines, we scan to determine where to split this partition. By visual inspection, we find that splitting on an annual income of $20,000 gives us the best split. Of the eight customers who borrowed $40,000 or less and earn more than $20,000 a year, seven paid their loan back in full and one defaulted. Because not default is the dominant outcome in this partition, we generalize that any future customers who earn more than $20,000 a year and borrow $40,000 or less will also pay back their loan in full. In similar fashion, we generalize that any future customers who earn less than $20,000 a year and borrow $40,000 or less will default on their loan. Note that each of the partitions now only have one misclassified example. They are much purer. These two new partitions and the generalizations we made for them result in a structural change to our classification tree. The tree will now include a new decision node which branches into two new leaf nodes. We can continue the recursive partitioning process in an attempt to create smaller and more homogenous partitions, or we can stop here. Generally, classification tree algorithms continue the recursive partitioning process until all of the instances within the partition are of the same class or value, or all the features in the dataset have been exhausted, or when some user-defined condition has been satisfied.

### **How do classification trees measure impurity?**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] Classification trees are built using a process known as recursive partitioning. The objective of recursive partitioning is to create child partitions that are purer than their parents. Classification tree algorithms use a mathematical formula to quantify the degree of impurity within a partition. Two of the most commonly used measures of impurity are entropy and Gini. Entropy is the preferred measure of impurity for the C5.0 decision tree algorithm. It is a concept that is borrowed from information theory. And when applied to decision trees, represents a quantification of the level of randomness or disorder that exists within a partition. A partition with low entropy is one in which most of the items have the same value or outcome, while a partition with high entropy is one that has no dominant outcome. The entropy of a partition S, with C possible outcomes or labels, is calculated as shown here. To understand how this formula works, let's walk through an example. Given the following orange partition with two possible outcomes, red triangle and green circle. To calculate the entropy of the partition, we first multiply the negative proportion of examples that belong to the triangle class by the binary log of the proportion itself. How do we get 0.53? Well, I'm glad you asked. There are 30 examples in the partition, and 16 of them belong to the triangle class. Therefore, the proportion of examples in the triangle class is 16 divided by 30, which is 0.53. Next, we multiply the negative proportion of examples that belong to the circle class by the binary log of the proportion itself. And how do we get 0.47? 14 out of 30 examples belong to the circle class. 0.47 is 14 divided by 30. Adding the two products yields 0.997. This is the entropy of the orange partition. So how does the classification tree algorithm use entropy to decide where to split the data during the recursive partitioning process? Let's assume that a classification tree algorithm is considering where to split this data in order to create pure partitions. One of the choices it has to make is whether a split on loan amount of $40,000 is the best split it could make. To figure this out, the algorithm starts by calculating the entropy of the left partition, which yields 0.986. Then it calculates the entropy of the right partition, which yields 0.951. The gray partition accounts for 47% of the data, while the yellow partition accounts for 53%. The combined entropy of both partitions after the split is a weighted sum of their individual entropy values, which is 0.47, times the entropy of the left partition, 0.986, plus 0.53 times the entropy of the right partition, 0.951. This comes out to 0.967. 0.967 is the combined entropy of the left and right partitions after the data is split on loan amount of $40,000. Notice that the entropy of the orange partition prior to the split is 0.997. And that the combined entropy of the gray and yellow partition is 0.967. This reduction in entropy that occurs as a result of the split is known as information gain. It is the difference between the entropy of the partition before the split and the combined entropy of the partitions after the split, which is 0.03. So what does all of this mean? Well, imagine that a classification tree algorithm is considering different split values for loan amount, as shown here. It'll first calculate the information gain that occurs as a result of each potential split. Then it'll choose the split that results in the largest information gain or reduction in entropy. That is the best split. This example uses information gained by way of entropy to determine the best split. As previously mentioned, entropy is a preferred impurity measure for the C5.0 decision tree algorithm. Another common impurity measure is Gini. It is a measure of statistical dispersion. Gini is the preferred measure of impurity for the CART decision tree algorithm. One way to think of Gini is as a measure of how often a particular example in a partition would be incorrectly labeled if it were randomly labeled based on the distribution of labels in the partition. Similar to entropy, the greater the degree of randomness or impurity within the partition, the higher the Gini impurity value. And the smaller the degree of randomness or impurity within a partition, the lower the Gini impurity value. The Gini impurity measure of a data partition S, with C possible outcomes or labels, is calculated as shown here. Similar to entropy, a classification tree algorithm that uses Gini as a measure of impurity will evaluate several splits, and choose the one that results in the largest reduction in Gini.

### **How is a regression tree built?**

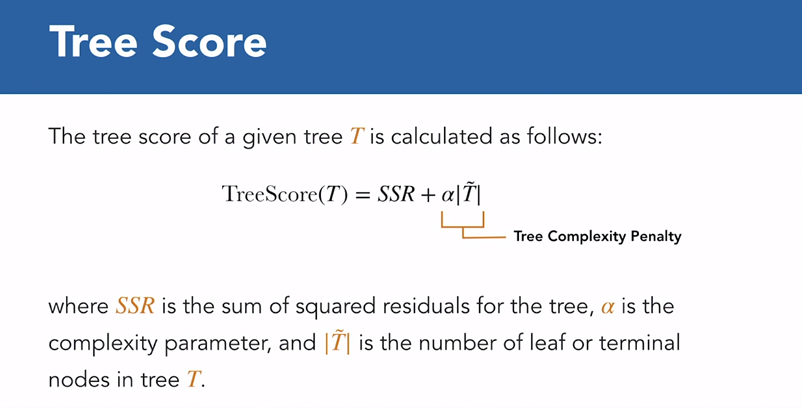
Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] Similar to classification trees, regression trees are built using a process known as recursive partitioning. For regression trees, the objective of recursive partitioning is to create successive child partitions that have less variability than their parent. To illustrate how recursive partitioning helps us build a regression tree, let's imagine that we work for a placement agency and that we just received the results of an income survey conducted by our agency. Each survey response includes the age of the worker, their level of education or highest degree earned, and their annual salary. Note that age and education level are the independent variables, or predictors, while salary is the dependent variable. Each of the survey responses can be represented on the scatter plot this way in terms of the dependent variable, annual salary, and one of the independent variables, age. Recall that for regression trees, the idea behind recursive partitioning is to repeatedly split data into smaller subsets in such a way that minimizes the variability of values within each subset. So the first thing a regression tree algorithm does is figure out how best to split this data into two so we have partitions or subsets that minimize variability the most. One of the measures that regression tree algorithms rely on to figure out the best split is the sum of squared residuals, or SSR. A residual is the difference between an observed data point and a reference data point such as the mean. The formula for computing the SSR of a partition with n values is shown here where Y-hat is the mean of the values in the partition and Y-i is each value in the partition. The SSR of a partition quantifies the overall difference between the values in the partition and the average of the values in the partition. A partition with high SSR implies that the values in the partition are dissimilar or very different from the mean. This is a partition that poorly explains the data. A partition with low SSR implies that the values in the partition are similar or close to the mean. This is a partition that explains the data well. So how does a regression tree algorithm use SSR to determine the best split? Well, I'm glad you asked. Let's assume that the first split the algorithm evaluates is where age is equal to 27.5. This is the halfway point between the data points for age 25 and those for age 30. The values in the left partition are 16.8, 43.9, and 50.4. The average of these values is 37. Recall that a residual is the difference between an observed data point and a reference data point. The reference data point in this example is the average value. So the residuals are the differences between each value and the mean. To get the SSR, we square the residuals and add them. This comes out to 635.2. Using the same approach for the right partition, we get an SSR of 13106.9. They combined some of squared residuals for both partitions if the data was split by age equal to 27.5 is the sum of the left SSR and the right SSR, which is 13742.1. In order to determine the split that reduces variability the most, the regression tree algorithm evaluates the SSR based on each possible split, and chooses the one with the lowest SSR, which is the split where age is equal to 40. This initial split creates the logic for the root node of our regression tree, which is shown here. It asks the question, is a worker 40 years old or younger? To create the branches and the next set of nodes, the regression tree algorithm makes some generalizations or simplifying assumptions based on the data in the two partitions. The first generalization it makes is based on the left partition. It's estimates that if a worker is 40 years old or younger, then the annual salary will be 44,503, which is the average of the left partition. The second generalization the regression tree algorithm makes is based on the data in the right partition. It estimates that if a worker is older than 40, then their annual salary will be 77,990, which is the average of the right partition. Depending on the data or some user defined criteria, the regression tree algorithm could stop the recursive partitioning process here, or it could keep partitioning the data into smaller subsets with less variability. If the algorithm continues with the recursive partitioning process, the structure of the regression tree will continue to evolve as more partitions are created within the data.

### **How to prune a decision tree**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] The basic idea behind recursive partitioning is to repeatedly split data into smaller subsets in such a way that maximizes the homogeneity or similarity of items within each subset. Generally, this process continues until all of the instances within a partition are of the same class or value or all the features in the dataset have been exhausted or when some user-defined condition has been satisfied. Usually, if we allow a tree to grow uninhibited until it meets the first two criteria, it may be too large or it may overfit against the training data. Overfitting occurs when a decision tree fits our data too perfectly. A tree that overfits does a great job explaining the data in the training set but performs poorly when given new or test data. To avoid overfitting, we have to carefully manage the size of a decision tree during or after the recursive partitioning process. this process is known as pruning. Pruning helps a decision tree to generalize well, which means that it will likely perform well when presented with previously unseen data. We can limit the size of a decision tree during the recursive partitioning process by setting criteria that need to be met at each split point. These criteria can be in the form of the maximum number of features to be considered for each split, the maximum number of decision nodes to allow for the tree or the minimum number of data points to allow in each partition. This approach to pruning is known as pre-pruning. Pre-pruning is appealing in that prevents unnecessary branches and nodes from being created, which saves compute cycles. However, pre-pruning could stop tree growth too early. This means that the tree may not get a chance to learn certain patterns in the data. An alternative pruning approach is post-pruning. As the name suggests, the idea here is to allow the decision tree to grow as large as it can and then reduce its size afterward. With regards to compute time, post-pruning is not as sufficient as pre-pruning, however, it does provide the benefits of being more effective in discovering important patterns within the data. There are several approaches to post-pruning. One of the most popular is known as cost complexity pruning or weakest link pruning. To help explain our cost complexity pruning works, let's imagine that we work for a placement agency and that we just received the results of an income survey conducted by our agency. Each survey response includes the age of the worker, their level of education or highest degree earned and their annual salary. Limiting our focus to age and annual salary, each of the survey responses can be represented on a scatterplot this way. Recall that the idea behind recursive partitioning is to repeatedly split data into smaller subsets in such a way that maximizes the homogeneity or similarity of items within each subset. If we allow the recursive partitioning process to continue uninhibited on this data, it would partition on every possible age value and we would end up with partitions that look like this. This would result in a decision tree that looked like this. This decision tree fits our data too perfectly and has likely overfit. A better decision tree for our data is one that is based on fewer than the maximum number of partitions, like one of these shown here. We can think of these sub-trees of pruned version of the previous one. The question then is how do we choose the best sub-tree to use? This is where cost complexity pruning comes in. The first step in the process is to calculate the sum of squared residuals or SSR for each sub-tree. For more information on how to compute the SSR for a tree, watch the previous video in this course sequence, titled "How is a regression tree built?" Notice that the larger the tree gets, the lower its SSR. This means that the larger the tree is, the better job it does at explaining the relationship between the independent variables and the dependent variable. However, we also know that the larger the tree is, the more likely it is to overfit against our data. To account for this, a better metric for which sub-tree to choose is the tree score or cost complexity measure. **The cost complexity measure for a tree is the sum of squared residuals for the tree plus the tree complexity penalty. The tree complexity penalty compensates for the number of leaf nodes in the tree. It is a product of the complexity parameter, a user-defined parameter and the number of leaf nodes in the tree.** As a side note, because the decision tree used in this illustration is a regression tree, we use the sum of squared residuals in calculating the tree score. If we were working with a classification tree, we would use entropy or gini instead of SSR to calculate the tree score. **If we set the alpha, the complexity parameter, to 1,000, the tree score for each sub-tree will be as shown here. This means that we will choose the first sub-tree because it has the lowest tree score. As you may have noticed, the choice of alpha has a significant impact on which tree ends up having the lowest tree score.** For example, if we reduce the value of alpha to 100, we get different tree scores for each tree. This means that we would choose this sub-tree instead because it now has the lowest tree score. Higher values for alpha favor simpler trees, while smaller values for alpha favor more complex trees. The question then is how do we decide on a value for alpha? The simple answer is that alpha is a hyperparameter and we use a process known as hyperparameter tuning to find the best value. Conceptually, finding the best alpha value involves training several sub-trees based on different values for alpha, then choosing the sub-tree that performs the best on the test data. The alpha for this tree is the best value for alpha.



### **Why and when to use a decision tree**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Man] In order to know when to use a decision tree, we need to understand its strengths and weaknesses. In terms of strengths, decision trees are simple to understand and interpret. The logical structure of a tree is intuitive, and easy to follow. This means we can easily create business rules, based on the structure of a tree. Unlike some other approaches, which work better with either discreet, or continuous features, decision trees are able to handle both types of data, very well. In other words, decision trees are useful, for both classification and regression problems. Decision trees also do very well, in handling missing, noisy and outlier data. This means we don't need to pre-process our data much, before we can use it to build a decision tree model. During each stage of the recursive partitioning process, the feature that reduces impurity the most is chosen. This means that unimportant features, are ignored by the tree. As a result, feature selection is not necessary, when working with decision trees. Decision trees do well on most problems, even if it makes slightly erroneous assumptions, about the nature of the data that is used to build it. Decision trees are useful on both small and large data sets. However, similar to other non-parametric models, the predictive accuracy of a decision tree, tends to improve as it encounters more training examples. There are some weaknesses inherent with decision trees, as well. The recursive partitioning process tends to be biased, toward features with a large number of unique values, when using entropy as a measure of impurity. Decision trees can be rather unstable, because small changes in data can result in large changes, in the structure of the tree. This limitation is mitigated by using decision trees, as part of an ensemble. Decision trees are non-parametric models. This means that they make very few assumptions, about the data. A decision tree attempts to describe the training data, as closely as possible. If not properly remediated, they can easily over fit against the training data. Decision trees can also under fit, if the pruning process is overly aggressive. Decision trees are limited to axis-parallel splits. This means that you can only split data, horizontally or vertically, during the recursive partitioning process. This limits the usefulness of decision trees, in certain problem domains. While decision trees are easy to understand, very large trees can be rather difficult to interpret. Decision trees tend to be biased, if the training data suffers from class imbalance. To mitigate this, we often have to balance the training data, before trying to fit a decision tree to it.

## **Question 1 of 11**

A partition with high SSR (Sum of Squared Residuals) is a partition with low variability.

* TRUE  
  Incorrect  
  A partition with high SSR implies that the values in the partition are dissimilar (or very different from the mean). This is a partition that poorly explains the data.
* FALSE



Replay

Review this video

How is a regression tree built?

5m 40s

## **Question 2 of 11**

If you allow a decision tree to grow as large as it can, and then reduce its size afterwards, what are you doing?

* pre-pruning
* downsizing
* post-pruning
* underfitting  
  Incorrect



Replay

Review this video

How to prune a decision tree

7m 10s

## **Question 3 of 11**

The preferred measure of impurity for the CART decision tree algorithm is Gini.

* FALSE  
  Incorrect  
  Gini is the preferred measure of impurity for the CART decision tree algorithm.
* TRUE



Replay

Review this video

How do classification trees measure impurity?

5m 48s

## **Question 4 of 11**

The reduction in entropy that occurs as a result of a split is known as \_\_\_\_\_.

* information gain
* gain index
* Gini  
  Incorrect
* combined entropy



Replay

Review this video

How do classification trees measure impurity?

5m 48s

## **Question 5 of 11**

Decision trees are built using a process that involves repeatedly splitting data into smaller and more homogenous subsets. What is this process called?

* information gain
* similarity  
  Incorrect
* homogeneity
* recursive partitioning



Replay

Review this video

How is a classification tree built?

6m 27s

## **Question 6 of 11**

What is not one of the stopping criteria for the recursive partitioning process?

* All nodes have been created.
* All the features in the dataset have been exhausted.
* A user-defined condition has been satisfied.  
  Incorrect
* All of the instances within a partition are of the same class or value.



Replay

Review this video

How is a classification tree built?

6m 27s

## **Question 7 of 11**

What is not a weakness of decision trees?

* Decision trees are only useful on a limited set of problems.
* Decision trees tend to be biased if the training data suffers from class imbalance.
* Decision trees are limited to axis-parallel splits.  
  Incorrect
* Decision trees can be rather unstable because small changes in data can result in large changes in the structure of the tree.



Replay

Review this video

Why and when to use a decision tree

2m 59s

## **Question 8 of 11**

A decision tree is an unsupervised machine learning approach.

* FALSE  
  Correct  
  Decision trees are a supervised machine learning approach. This means that you can use a decision tree to predict future outcomes.
* TRUE

## **Question 9 of 11**

What could happen to a decision tree if you don't properly manage its size?

* It could overfit.
* It could underfit.
* It could grow forever.  
  Incorrect
* It could become too robust.



Replay

Review this video

How to prune a decision tree

7m 10s

## **Question 10 of 11**

What type of decision tree should you build in order to predict the salary of a worker based on their age and work experience?

* prediction tree  
  Incorrect
* classification tree
* regression tree
* supervised tree



Replay

Review this video

What is a decision tree?

4m 29s

## **Question 11 of 11**

What is not a strength of decision trees?

* Decision trees do very well in handling missing, noisy, and outlier data.
* Very large decision trees are easy to understand.  
  Correct  
  One of the weaknesses of decision trees is that while decision trees are easy to understand, very large trees can be rather difficult to interpret.
* Decision trees are useful on both small and large datasets.
* Decision trees are able to handle both continuous and categorical variables.

### **How to build a classification tree in Python**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] In this exercise, we'll use a sample loans data set to build a classification tree that predicts whether a borrower will default or not default on a new loan. Before we get started, note that this video is the first in a three video sequence, that explains how to build, visualize, and prune a classification tree. We start by importing the Pandas package. Then we import the data into a data frame called loan and preview it to make sure that the input worked as expected. Now that we have our data, let's try to understand it. First, we get a concise summary of the structure of the data by calling the info method of the data frame. From the summary, we can tell that there are 30 instances in the dataset by looking at the range index. We can also tell that there are three features in the dataset. Looking at the D type column of the summary, we see that the income and loan amount columns hold integer values while the default column holds text, AKA object. Next, we get summary statistics for the data by calling the described method of the data frame. From the statistics, we see that the minimum income value in the data is five. While the maximum value is 34. Note that these values are in the thousands. So what we're seeing here is $5,000 and $34,000. Likewise, the average loan amount is $51,967. Next, let's also visually explore the data by creating a few plots. To ensure that our plots show up in line, we run the map plot lib in line command. Then, we import two packages, mapplotlib pyplot and the seaborne package. The first plot we create is a boxplot. This is a plot to show the difference in annual income between those that did not default, which is no, and those that did default, which is yes The plot shows that those that did not default on their loans tend to have a higher annual income. Next, let's create another boxplot to show the difference in loan amount between those that did not default on their loans and those that did. This chart shows that those that defaulted on their loans tend to have borrowed a little slightly more than those that did not. Finally, let's create a scatter plot to look at the relationship between income and loan amount. This chart doesn't show a clear linear relationship between those two variables. There isn't much we can really infer from it, so we can move on now. Before we build our classification tree though, we need to split the data into training and test sets. Prior to doing so, we must first separate the dependent variable from the independent variables. Let's start by creating a data frame called Y for the dependent variable, which is default. Then we also do the same and create a second data frame called X for the independent variables, income and loan amount. Now that we have our two data frames, we need to now build our model. We can now split our data, before we do so, we have to import the train test split function from the SK learn model selections package. Using this, we can split the data, the X and Y data frames, into X\_train X\_test, Y\_train and Y\_test. Note that here we set train size to 0.8. This means we want 80% of the original data to become the training data, while 20% becomes the test data. We also set stratify as y which means we want the data splits using stratified random sampling based on the values of y. Finally, we set random state to 1234, simply so we get the same results every time we do the split. Now that our data is split, the shape attribute of the X\_train and X\_test data frames tell us how many instances, or records, are in each data frame. We can see that we have 24 instances in the training set and six instances in the test set. To build a classification tree in Python, we need to import the decision tree classifier class from the SK learn tree sub package. We then instant sheet an object from the class. We call the object classifier. Now that we have an object, we can build or fit a classification tree model using the training data. To evaluate and estimate the future performance of our model, we can now see how this model fits against the test data. To do so, we pass the test data to the score method of the model. This returns the accuracy of the model against the test data. Hmm, a classification tree is only able to accurately explain 50% of the relationship between the independent variables and a dependent variable within the test data. That's no better than a coin toss. We can do better. Over the course of the next two videos, we will visualize this tree then attempt to improve its performance. I'll see you there.

### **How to visualize a classification tree in Python**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] Before we get started, note that this video is a second in a three video sequence that explains how to build, visualize, and prune a classification tree. So if you have not done so, watch the previous video for a detailed explanation of the prior code. Now that we've trained a classification tree, let's visualize it to get a better understanding of the tree logic. First, we make sure that we import the tree object from the sklearn package. The figure\_method of Pyplot allows us to specify the size of our tree. Feel free to adjust this to see how it impacts the size of your tree. Finally, we use the plot\_tree method of the tree object to visualize the tree. The first argument we pass to this method is the classification tree model itself, model. Then we specify the independent variables as a list. Next, we specify the possible values of the dependent variable as a list in ascending order, No and Yes. Finally, we specify that we want the nodes of the Tree color filled. Let's run the code. Now we have our tree. Let's take some time to understand the structure of this classification tree. We see that the root node asks the question is income less than or equal to $14,500? This means that the first splits that the classifier made during the recursive partitioning process is that income equal to 14.5. The fact that income variable was used as the first split, let's us know that it is the most important variable within the dataset in predicting the outcome. The branch to the left of each node is for the Yes response, while the branch to the right is for the No response. Within each node, we get a value for the Gini impurity score. Gini is a measure of the degree of impurity in the partition. The smaller this value is, the more homogenous the items in a partition are. We also see the number of items or samples within each partition. Notice that this value decreases as we work our way down the tree towards the leaf nodes. This is expected since the primary objective of recursive partitioning is to create smaller, more homogenous subsets of the data. The next information in each node, value, indicates the count of items within each class. This is the item distribution. For example, in the root node there are 14 items with a value of No and 10 with a value of Yes. The Noes are the majority, which is why the class value is equal to No. This means that if our classification tree were just one node, the root node, it would label every loan as not default. Notice how the Gini impurity values change in relation to the item distributions. As one class dominates, the Gini value tends toward zero. One of the benefits of decision trees is that they are pretty good at ranking the effectiveness of independent variables and predicting the values of the dependent variable. This is known as feature\_importance. We can visualize the feature\_importance of the independent variables as follows. First, we assign the feature importances on the score attribute of the model to a variable, which we call Importance. The attribute returns an array of the important scores of each independent variable. Next, we create a Pandas Series called feature\_importance by using the importance array as the values and the independent variable names as the index. Finally, we plot the series. Let's take a look at it. From the plot, we see that the income variable is more important than the loan amount in predicting whether a borrower will default on their loan or not.

### **How to prune a classification tree in Python**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] Before we get started, know that this video is a third in a three video sequence. That explains how to build, visualize and prune a classification tree. So if you have not done so, watch the previous two videos for a detailed explanation of the prior code. Now that we've trained and visualized a classification tree, let's look into what we can do to improve its performance by pruning. Decision trees are prone to overfitting. One telltale sign that a tree has overfit is if it has a high accuracy score on the training data with a low accuracy score on the test data. Let's start by getting our trees accuracy on the training data. To do this we pass the training data to the score method of the model. A model is a hundred percent accurate on a training data. That's suspicious. Let's get a second opinion from the test data. Similarly, we pass the test data to the score method of the model. Our model is 50% accurate on the test data. Our model has definitely overfit on the training data and needs to be pruned. There are two ways to prune a decision tree. One is to set parameters that manage its growth during the recursive partitioning process. This is known as pre-pruning. Another approach is to allow the tree to fully grow on impeded and then gradually reduce its size in order to improve its performance. This is known as post-pruning. In this tutorial, we will use a pre-running approach. This means that we need to figure out the best combination of values for the parameters of the tree that will result in the best performance. This is known as hyper parameter tuning. The psyche learned package scikit-learn provides several parameters we can tune during this process. We will limit ourselves to three of them. We start by creating a dictionary which we call grid that holds the values of the parameters we want to try out. The first parameter is max depth. This sets the maximum depth of the decision tree. We will try setting the value to two, three, four and five to see which is the best. The next parameter is min sample split. This sets the minimum number of items we can have in the partition before it can be split. Studies show that a value between one and 40 is best. We will try setting the value to two, three, and four. Next is the min samples leaf parameter. This sets the minimum number of items we have in a leaf node. Studies show that the best values are between one and 20. We will try setting the value to one, two, three, four, five, and six. The gridsearch CV class from the scikit-learn model selection sub package allows us to perform a great search to find the best parameter values for our tree. We import the class then we instantiate a decision tree classifier object and then we pass the object to a new grid search CV object, which we call GCV. We also pass the parameter grid to the object. We then pass the training data to the fit method of GCV so it evaluates each hyper parameter combination in grid. The best estimator attributes of GCV returns the classifier with the best combination of hyper parameters for our data, let's get that. We then fit a classification tree on the training data using this classifier. The output shows that the best combination of hyper parameters is max depth set at two and min samples leaf set at six. Now we can reevaluate how well our model fits the training data by passing the training data to the score method of the model. We see that the accuracy has gone down from a hundred percent to 87.5%. Let's see how the model fits the test data as well. Now, the model's accuracy on the test data has risen from 50% to 83.3% that is much better. Finally, we can visualize our prune model. Our prune tree is much smaller than the one we started off with but it generalizes much better.

### **How to build a regression tree in Python**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] In this exercise, we'll use a sample income data set to build a regression tree that predicts the salary of a worker based on their age and education level. Before we get started, know that this video is the first in the three video sequence that explains how to build, visualize, and prune a regression tree. We start by importing the Pandas package. Then we import the data into a data frame called income and preview it to make sure that the import worked as expected. Now that we have our data, let's try to understand it. First, we get a concise summary of the structure of the data by calling the info method of the data frame. From the summary, we can tell that there are 30 instances in the data set by looking at the range index. We can also tell that there are three features in the data set. Looking at the D type column of the summary, we see that the age column holds integer values. The education column holds text, AKA object, and the salary column, holds floating point or decimal values. Next, we get summary statistics for the numeric columns by calling the described method of the data frame. From the statistics, we see that the minimum salary value in the data is 16.8 while the maximum value is 118. Note that these values are in the thousands. So what we're seeing here, is $16,800 and $118,000. We also see that the minimum median and maximum age values are 24, 45, and 65 respectively. Next, let's also visually explore the data by creating a few plots. To ensure that our plots show up in line, we run the map plot lib in line command. Then we import pie plot from the map plot lib package as well as a seaborne package. The first plot we create is a box plot that shows the distribution of salary by education level. The chart shows that those with a high school diploma tend to earn the least while those with a professional degree, tend to earn the most. Next let's create another box plot to show the distribution of age by education level. This chart doesn't show much separation between the groups. However, we do see that those with professional degrees tend to be a bit older than the rest of the workers in the data set. Finally, let's create a scatter plot to look at the relationship between salary and age. The chart shows somewhat of a linear relationship between these two variables. This means that generally, the older a worker is, the higher their salary. Now that we've done some initial data exploration let's prepare our data for modeling by splitting it into training and test sets. Prior to doing so, we must first separate the dependent variable from the independent variables. Let's start by creating a data frame called Y for the dependent variable, which is salary. Then we create a second data frame X, for the independent variables, age and education. Next, we import the train test split function from the SK learn model selections package. Then we split the X and Y data frames into X\_train X\_test Y train and Y test. Note that we set train size to 0.6. This means we want 60% of the original data to become the training data. While 40% becomes the test data. We also set stratify to the education column in X, which means we want the data split using stratified random sampling based on the values of the education column. Finally, we set random state to 1234. Simply so we get the same results every time we do the split. The shape attribute of the X\_train and X\_test data frames tell us how many instances or records are in each data set. From the results, we can see that we have 18 instances in the training set and 12 instances in the test set. The psyche learn package we intend to use to fill out regression trees does not support non-numeric values, like the education column in our data. As a result, we have to dummy code the education columns in the X\_train and X\_test data frames. Before we dummy code X\_train let's preview it using the head method. To dummy code X\_train, we pass it to the Pandas get dummy function and preview the updated data frame. Notice that each of the categorical values in the education column are now columns, each with a dichotomous value of zero and one. Let's dummy code and preview the test data set X\_test as well. We are done with data preparation. We can now build our model. To build the regression tree in Python, we need to import the decision tree regressor class from the SK learn tree sub package. We then instantiate an object from the class. We call the object, regressor. Using the regressor object, we can fit a regression tree on the training data. To evaluate and estimate the future performance of a model, let's see how well it fits against the test data. To do so, we pass the test data to the score method of the model. This returns the R squared of the model on the test data. The R squared value we get here tells us that our model is only able to explain 58.5% of the variability and the response values of the test data. We can do better. Another way to evaluate a regression tree is to evaluate how accurate it is. This means comparing the predicted values against actual values or getting the mean absolute error of the predictions. Before we can get the mean absolute error, we need to get the models predicted response values for the test data. We assign these results to a variable called y\_test\_pred. Next, we import the mean absolute error function from the SK learn metrics of package and calculate the mean absolute error between the actual response values y\_test and the predicted response values y\_test\_pred. What does this mean? The mean absolute error implies that going forward, we should expect the salary values our regression tree predicts, to be off the mark by an average of plus or minus 13,542.

### **How to visualize a regression tree in Python**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] Before we get started, note that this video is the second in a three video sequence that explains how to build, visualize and prune a regression tree. So if you have not done so, watch the previous video for a detailed walkthrough of the code and the cells above. Now that we've trained on regression tree, let's visualize it to get a better understanding of the tree logic. First, we make sure that we import the tree object from the sklearn package. The figure method of Pyplot allows us to specify the size of our tree, feel free to adjust this to see how it impacts the size of your tree. Next, we'll use the plot tree method of the tree object to visualize the tree. The first argument we pass to this method is the regression tree model itself. Then we specify the dummy coded independent variables as a list. Finally, we specify that we want the nodes of the tree color field. Because of the depth, and number of nodes in our tree, it's a bit difficult to make out the details of the tree. For illustrative purposes alone, let's limit the tree visualization to the first three nodes in the tree. To do this, we set the max depth argument in the plot tree method to one. Now we have a clearer view of the top set of nodes. Let's take a moment to understand the structure of the tree based on what we have here. We can interpret the root node as asking the question, is a worker 34 years old or younger? If so, branch to the left, else branch to the right. The fact that the age variable was used as a first split, lets us know that it is the most important variable within the data set in predicting the salary of a worker. Know that the branch logic and the next two nodes are a bit peculiar. For example, the right branch evaluates whether education professional is less than or equal to 0.5. To understand what is happening here, recall that we had to dummy code original education values to either zero or one. So if a worker has a professional degree, the value will be one and zero if they don't. With that in mind, we can interpret this branch logic as asking the question, does a worker not have a professional degree? If yes, branch to the left, else branch to the right. I know, it does sound a bit odd. So we can ask the opposite question and reverse the direction of the responses. In other words, we can ask, does a worker have a professional degree, if so, branch through right, else branch to the left. Within each node, besides the branch logic, we also get a value for the MSE or mean squared error. This can be interpreted as a measure of the degree of impurity, or variability in a partition. The smaller this value is, the closer the values are to the mean. Conversely, the larger this value is, the further the values are to the mean. We also see the number of items or samples within each partition. Notice that this value decreases as we work our way down the tree towards the leaf nodes. This is expected since the primary objective of recursive partitioning is to create smaller, more homogenous subsets of the data. The last information in each node value is the mean of the values in a particular partition. This is the predicted value of the regression tree. For example, if the regression tree were just one node, the root node, the tree would predict that all worker salaries was $65,367. The average of all this error is in the training data. One of the benefits of decision trees is that they are pretty good at ranking the effectiveness of independent variables in predicting the values of the dependent variable. This is known as feature importance. We can get the feature importance of each independent variable from the feature importances on the score attributes of our model. We get back an array of important scores for each independent variable. To put these scores in context, let's connect them to the feature names and visualize the scores. To do this, the first thing we do is create a Pantha series called feature importance by using the importance array as the values and the independent variable names as the index. Then we sort the series by value and plot it. From the plot, we see that the age variable is the most important in predicting salary. While the education masters feature is the least important.

### **How to prune a regression tree in Python**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] Before we get started, note that this video is the third in a three-video sequence that explains how to build, visualize and prune a regression tree in Python. If you have not done so, watch the previous two videos for a detailed walkthrough of the code in the cells above. Now that we've trained and visualized a regression tree, let's look into what we can do to improve its performance by pruning. Decision trees are prone to overfitting. One telltale sign that a tree has overfit is if it performs well on the training data but very poorly on the test data. Let's evaluate our tree to see if it overfit on the training data. To do this, we pass the training data to the score method of the model. Our model is able to explain 99% of the variability in the training data. Let's see how it does on the test data. Similarly, we pass the test data to the score method of the model. Our model is only able to explain 59% of the variability in the test data. It has overfit the training data and needs to be pruned. There are two ways to prune a decision tree. One is to set parameters that manage its growth during the recursive partitioning process. This is known as pre-pruning. Another approach is to allow the tree to fully grow unimpeded and then gradually reduce its size in order to improve its performance. This is known as post-pruning. In this tutorial, we will use the post-pruning approach. The specific strategy we will use is known as cost complexity pruning. The primary objective in cost complexity pruning is finding the right parameter, known as alpha. The right alpha is the one that performs the best with the test data. To get a list of effective alpha values to choose from, we start by passing the training data to the cost complexity pruning path method of our previously instantiated regressor object. Then we extract a list of the effective alphas. The list of effective alphas go from zero all the way to 222.77. The larger the value for alpha, the smaller the tree will be. The maximum value of alpha represents a tree with just one node. We do not want that one. We remove it from our list of effective alphas. That's better. Next, we train and evaluate several trees using different values for alpha. We start by creating two empty lists: train scores and test scores to store the results of our model evaluation. Then we loop through all the alpha values in our list of effective alphas. For each alpha, we instantiate a regressor with the alpha, fit a regression tree to the training data, evaluate the tree's performance on the training and test datasets and append the results to the train scores and test scores list. Let's run that. With that information, we can now plot the training and test scores against different values of alpha. The plot shows that when alpha is zero, the tree overfits. The training score is at its highest. As alpha increases, more of the tree is pruned, which results in reduced training scores. The test scores behave a little differently. As alpha increases, the test score initially increases. Then it starts to decline as well. The best alpha is the one that corresponds with the highest test score. By a visual inspection alone, this seems to fall somewhere between 10 and 20. Let's get a list of these test scores. The test scores are listed in the same order as the effective alphas in the ccp\_alpha's list. To get the value for the best alpha, we first get the index of the highest test score and I use the index to select the corresponding alpha from the ccp\_alpha's list. We see that the best alpha for the regression tree is 14.8. Finally, we fit a regression tree on the training data pruned using the best alpha. Let's get the model's R squared on the training data. We see that the R squared on the training data has gone down from .99 to .877. Now, let's get the model's R squared on the test data as well. We see that the R squared on the test data has gone up from .585 to .757. Finally, we can visualize our pruned regression tree. Our new regression tree is smaller on the one we started off with but it performs better on the test data, which means that it now generalizes better.

### **Next steps with decision trees**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Frederick] Congratulations. You now know what a decision tree is, how it's built, and when to use one. You've learned how to build, visualize, and prune a classification tree, as well as a regression tree in Python. The foundational knowledge and skills you've acquired in this course should serve as a stepping stone to continue learning about machine learning. Specifically, it should serve as a launchpad for solving more complex, supervised machine learning problems using decision trees. Here are a few recommended next steps. Decision trees are one of many supervised machine learning models we can build in Python. I encourage you to continue to explore other LinkedIn Learning courses that illustrate the use of different types of machine learning models. One such course is Machine Learning with Python, k-Means Clustering. Besides courses that explore other models, I also encourage you to explore courses that highlight the importance of ethics in data collection and use. An example of such a course is Data Ethics, Watching Out for Data Misuse. If you're interested in broadening your skillset into other languages, such as R, then also check out my book, "Practical Machine Learning in R." Finally, I recommend that you continue to practice what you've learned. Find new problems to solve, find interest in data sets on which to practice your new skills. The journey into the world of machine learning with Python is a lifelong one. Thanks for coming along with me on this journey. I'll see you next time.