Photo by AbsolutVision on Unsplash

Entropy and Information Gain in Decision Trees

A simple look at some key Information Theory concepts and how to use them when building a Decision Tree Algorithm.

Jeremiah Lutes Nov 16, 2020 · 12 min read ★

> *What criteria should a decision tree algorithm use to split variables/columns?*

Before building a decision tree algorithm the first step is to answer this question. Let's take a look at one of the ways to answer this question. To do so we will need to understand a use a few key concepts from information theory.

Let's examine this method by taking the following steps:

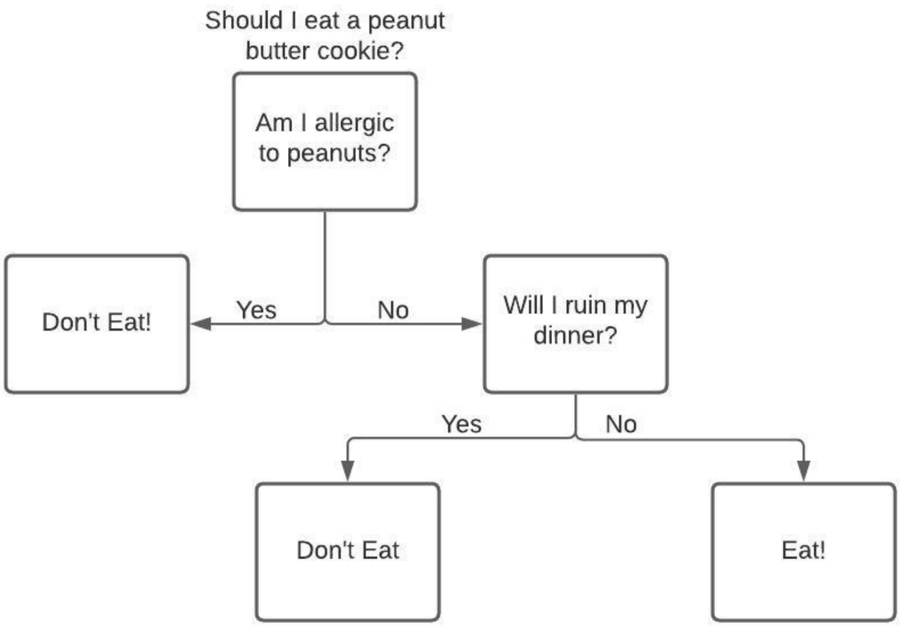
1. Take a very brief look at what a **Decision Tree** is.
2. Define and examine the formula for **Entropy.**
3. Discuss what a **Bit** is in information theory.

4. Define **Information Gain** and use entropy to calculate it.
5. Write some basic **Python functions** using the above concepts.

Decision Trees

In data science, the decision tree algorithm is a supervised learning algorithm for classification or regression problems. Our end goal is to use historical data to predict an outcome. Unlike linear regression, decision trees can pick up nonlinear interactions between variables in the data.

Let's look at a very simple decision tree. Below is a workflow that can be used to make a decision on whether or not to eat a peanut butter cookie.

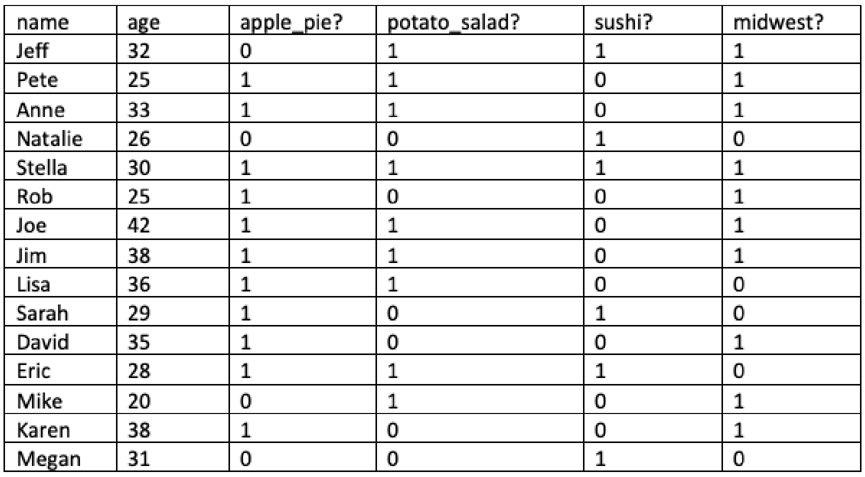

Should I eat a peanut butter cookie?
Am I allergic to peanuts?
Yes
No
Don't Eat!
Will I ruin my dinner?
Yes
No
Don't Eat
Eat!

A decision tree example on whether or not to eat a cookie

In this example, a decision tree can pick up on the fact that you should only eat the cookie if certain criteria are met. This is the ultimate goal of a decision tree. We want to keep making decisions(splits) until certain criteria are met. Once met we can use it to classify or make a prediction. This example is very basic using only two variables (allergy, ruining dinner). But, if you have a dataset with thousands of variables/columns how do you decide which variables/columns are the most efficient to split on? A popular way to solve this problem, especially if using an ID3 algorithm, is to use **entropy** and **information gain**.

The Task

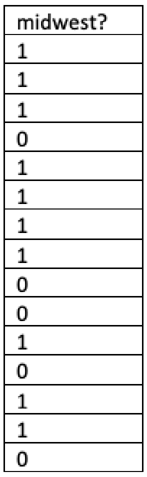
Let's say we have some data and we want to use it to make an online quiz that predicts something about the quiz taker. After looking at the relationships in the data we have decided to use a decision tree algorithm. *If you have never been sucked into an online quiz, you can see hundreds of examples here.* The goal of the quiz will be to guess if the quiz taker is from one of America’s midwest states. The questions in the quiz will revolve around if they like a certain type of food or not. Below has a small fictional dataset with fifteen entries. Each entry has answers to a series of questions. Most questions are about if they liked a certain type of food, in which the participant answered (1) for yes or (0) for now. The last column(“midwest?”) is our **target column,** meaning that once the decision tree is built, this is the classification we are trying to guess.



name	age	apple_pie?	potato_salad?	sushi?	midwest?
Jeff	32	0	1	1	1
Pete	25	1	1	0	1
Anne	33	1	1	0	1
Natalie	26	0	0	1	0
Stella	30	1	1	1	1
Rob	25	1	0	0	1
Joe	42	1	1	0	1
Jim	38	1	1	0	1
Lisa	36	1	1	0	0
Sarah	29	1	0	1	0
David	35	1	0	0	1
Eric	28	1	1	1	0
Mike	20	0	1	0	1
Karen	38	1	0	0	1
Megan	31	0	0	1	0

Entropy

To get us started we will use an information theory metric called entropy. In data science, entropy is used as a way to measure how “mixed” a column is. Specifically, entropy is used to measure disorder. Let’s start by finding the entropy of our target column, “midwest?”.



midwest?
1
1
1
0
1
1
1
1
0
0
1
0
1
1
0

Our target column, “midwest?”

There are ten people who live in the midwest and five people who don’t. If someone was going to ask you how mixed the column is, you could say it was sort of mixed, with a majority(2/3) of the people from the midwest. Entropy gives us a way to quantify the answer” sort of mixed”. The more mixed the (1)s and (0)s in the column are, the higher the entropy. If “midwest?” had equal amounts of (1)s and (0)s our entropy would be 1. If “midwest?” consisted only of (1)s the entropy would be 0.

We can use the following formula to calculate entropy:

$$-\sum_{i=1}^{c} P(x_i)\log_b P(x_i)$$

the formula for entropy

Let's go through each step of the formula and calculate the entropy for the "midwest?" column.

1. We need to iterate through each unique value in a single column and assign it to i. For this example, we have 2 cases(c) in the "midwest?" column, either (0) or (1).
2. We then compute the probability of that value occurring in the data. For the case of (1), the probability is **10/15**. For the case of (0), the probability is **5/15**.
3. We take the probability of each case and multiply it by the logarithm base 2 of the probability. *2 is the most common base because entropy is measured in bits(more on that later). The full explanation of why 2 is used is out of the scope of this post, but a user on stack exchange offers a good explanation*. For the case of(1), we get **10/15*log2(10/15)**. For the case of (0), we get **5/15*log2(5/15).**
4. Next, we take our product from each case above and sum it together. For this example, **10/15*log2(10/15) + 5/15*log2(5/15).**
5. Finally, we negate the total sum from above, **— (10/15*log2(10/15) + 5/15*log2(5/15)).**

Once we put the steps all together we get the below:

$$\begin{gathered} -(10/15 \cdot \log_2(10/15) + 5/15 \cdot \log_2(5/15)) \\ -(-.389975 + -.528308) \\ -(-.918278) \\ .918278 \end{gathered}$$

Our final entropy is .918278. So, what does that really mean?

Information Theory and a Bit of Information

Moving forward it will be important to understand the concept of bit. In information theory, a bit is thought of as a binary number representing 0 for no information and 1 for a full bit of information. We can represent a bit of information as a binary number because it either has the value (1) or (0). Suppose there's an equal probability of it raining tomorrow (1) or not raining(0). If I tell you that it will rain tomorrow, I've given you one bit of information.

We can also think of entropy as information. Suppose we have a loaded six-sided die which always lands on (3). Each time we roll the die, we know upfront that the result will be (3). We gain no new information by rolling the die, so entropy is 0. On the other hand, if the die is far and we roll a (3) there was a 1/6 chance in rolling the (3). Now we have gained information. Thus, rolling the die gives us one bit of information — which side the number landed on.

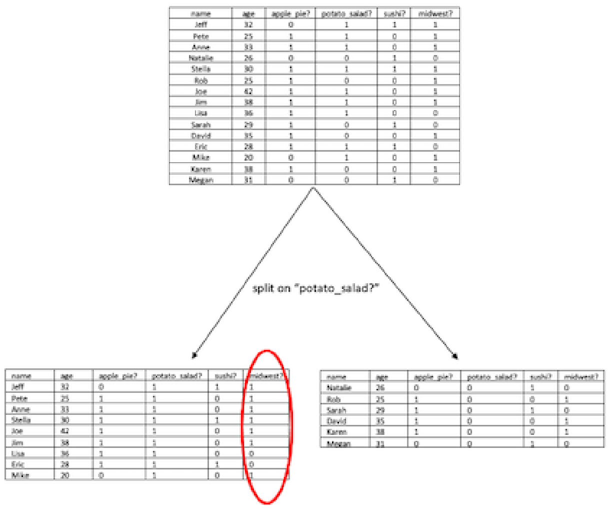
For a deeper dive into the concept of a bit of information, you can read more here.

We get less than one "bit" of information — only .918278 — because there are more (1)s in the "midwest?" column than (0)s. This means that if we were predicting a new value, we could guess that the answer is (1) and be right more often than wrong (because there's a 2/3 probability of the answer being 1). Due to this prior knowledge, we gain less than a full "bit" of information when we observe a new value.

Using Entropy to Make Decisions

Our goal is to find the best variable(s)/column(s) to split on when building a decision tree. Eventually, we want to keep splitting the variables/columns until our mixed target column is no longer mixed.

For instance, let's look at the entropy of the "midwest?" column after we have split our dataset on the "potato_salad?" column.



name	age	apple_pie?	potato_salad?	sushi?	midwest?
Jeff	32	0	1	1	1
Pete	25	1	1	0	1
Anne	33	1	1	0	1
Natalie	26	0	0	1	0
Stella	30	1	1	1	1
Rob	25	1	0	0	1
Joe	42	1	1	0	1
Jim	38	1	1	0	1
Lisa	36	1	1	0	0
Sarah	29	1	0	1	0
David	35	1	0	0	1
Eric	28	1	1	1	0
Mike	20	0	1	0	1
Karen	38	1	0	0	1
Megan	31	0	0	1	0

name	age	apple_pie?	potato_salad?	sushi?	midwest?
Jeff	32	0	1	1	1
Pete	25	1	1	0	1
Anne	33	1	1	0	1
Stella	30	1	1	1	1
Joe	42	1	1	0	1
Jim	38	1	1	0	1
Lisa	36	1	1	0	0
Eric	28	1	1	1	0
Mike	20	0	1	0	1

name	age	apple_pie?	potato_salad?	sushi?	midwest?
Natalie	26	0	0	1	0
Rob	25	1	0	0	1
Sarah	29	1	0	1	0
David	35	1	0	0	1
Karen	38	1	0	0	1
Megan	31	0	0	1	0

split on the "potato_salad?" column

Above, our dataset is split in two sections. On the left side, everyone who likes potato salad. On the right side everyone who doesn't. We fill focus on the left side which now has seven people from the midwest and two people who aren't. By using the formula for entropy on the left split midwest column the new entropy is .764204. This is great! Our goal is to lower the entropy and we went from .918278 to .764204. But, we can't stop there, if we look at the right column our entropy went up as there are an equal amount of (1)s and (0)s. What we need is a way to see how the entropy changes on both sides of the split. The formula for information gain will do that. It gives us a number to quantify how many bits of information we have gained each time we split our data.

Information Gain

Earlier we established we want splits that lower the entropy of our target column. When we split on “potato_salad?” we saw that entropy in the “midwest?” went down on the left side. Now we need to understand the total entropy lowered when we look at both sides of the split. Let’s take a look at information gain.

Information gain will use the following formula:

$$IG(T,A) = Entropy(T) - \sum_{v \epsilon A} \frac{|T_v|}{T} \cdot Entropy(T_v)$$

Let’s breakdown what is going here.

We’ll go back to our “potato_salad?” example. The variables in the above formula will represent the following:

- T = Target, our “midwest?” column
- A = the variable(column) we are testing, “potato_salad?”
- v = each value in A, each value in the “potato_salad?” column

1. First, we’ll calculate the orginal entropy for (T) before the split , **.918278**
2. Then, for each unique value (v) in variable (A), we compute the number of rows in which (A) takes on the value (v), and divide it by the total number of rows. For the “potato_salad?” column we get **9/15** for the unique value of (1) and **6/15** for the unique value of (0).
3. Next, we multiply the results by the entropy of the rows where (A) is (v). For the left split(split on 1 for “potato_salad?”) we get **9/15 * .764204**. For the right side of the split (split on 0 for “potato_salad?”) **we get 6/15 * 1.**
4. We add all of these subset products together, **9/14*.764204 + 6/15 = .8585224**.

5. We then subtract from the overall entropy to get information gain, **.918278 -.8585224 = .059754**

Our information gain is .059754. What does that tell us?

Here's an alternate explanation. We're finding the entropy of each set post-split, weighting it by the number of items in each split, then subtracting from the current entropy. **If the result is positive, we've lowered entropy with our split. The higher the result is, the more we've lowered entropy.**

We end up with .059754, which means that we gain .059754 bits of information by splitting our data set on the "potato_salad?" variable/column. Our information gain is low, but it's still positive which is because we lowered the entropy on the left side of the split.

Now we need to repeat this process for every column we are using. Instead of doing this by hand let's write some Python code.

Wrapping It All Up With Python

Now that we understand information gain, we need a way to repeat this process to find the variable/column with the largest information gain. To do this, we can create a few simple functions in Python.

Importing the Data

Let's turn our above table into a DataFrame using the Python pandas library. We will import pandas and use the read_csv() function to make a DataFrame named "midwest".

```
import pandas as pd
midwest = pd.read_csv('midwes.csv')
```

A Python Function for Entropy

For this function, we will need the NumPy library to use the bincount() function and the math module to use the log() function.

```
import numpy
import math
```

Next, we will define our function with one parameter. The argument given will be the series, list, or NumPy array in which we are trying to calculate the entropy.

```
def calc_entropy(column):
```

We will need to find the percentage of each case in the column. We can use the numpy.bincount() function for this. The return value is a NumPy array which will store the count of each unique value from the column that was passed as an argument.

```
counts = numpy.bincount(column)
```

We'll store the probabilities of each unique value by dividing the "counts" array by the length of the column.

```
probabilities  = counts / len(column)
```

We can then initialize a variable named "entropy" and set it to 0.

```
entropy = 0
```

Next, we can use a "for loop" to loop through each probability in our probabilities array and multiply it by the logarithm base 2 of probability using the math.log() function. Then, add each case to our stored entropy variable. **be sure to check your probability is great than 0 otherwise log(0) will return undefined*

```
for prob in probabilities:
    if prob > 0:
       endtropy += prob * math.log(prob,2)
```

Finally, we'll return our negated entropy variable.

```
return -entropy
```

All together now:

Great! Now we can build a function to calculate information gain.

A Python Function for Information Gain

We’ll need to define a function that will have three parameters, one for the entire dataset, one for the name of the column we want to split on, and one for the name of our target column.

```
def calc_information_gain(data, split_name, target_name):
```

Next, we can use the entropy function from earlier to calculate the original entropy of our target column.

```
orginal_entropy = calc_entropy(data[target_name])
```

Now we need to split our column.

**For this example we will only use the variables/columns with two unique. If you want to split on a variable/column such as “age”, there are several ways to do this. One way is to split on every unique value. Another way is to simplify the calculation of information gain and make splits simpler by not splitting for each unique value. Instead, the median is found for the variable/coumn being split on. Any rows where the value of the variable is below the median will go to the left branch, and the rest of the rows will go to the right branch. To compute information gain, we’ll only have to compute entropies for two subsets. We won’t be walking through this method but once the split on the median is performed the rest of steps would be the same as outlined below.*

Since the columns we are working with only have two unique values we will make a left split and a right split.

We’ll start by using the pandas.Series.unique() to give us an array of the unique values in the column

```
values = data[split_name].unique()
```

Next, we will create a left and right split using “values”.

```
left_split = data[data[split_name] == values[0]]
right_split = data[data[split_name] == values[1]]
```

Now we can initiate a variable to subtract from our original entropy.

```
to_subtract = 0
```

Then we’ll iterate through each subset created by our split, calculate the probability of the subset, and then add the product of the probability and the subsets target column’s entropy.

```
for subset in [left_split, right_split]:
    prob = (subset.shape[0] / data.shape[0])
    to_subtract += prob * calc_entropy(subset[target_name])
```

Finally, we can return the difference of to_subract being subtracted from the original entropy.

```
return original_entropy - to_subtract
```

The entire function is below.

A Python Function for the Highest Information Gain

Our final function will be one that will return the variable/column name with the highest information gain.

As mentioned earlier we are only using the columns with two unique values for this example. We'll store those column names in a list to use in the function. *To get to the point we'll hard code this for this example but in a large dataset, it would be best to write code to build this list dynamically based on the criteria we use to choose the columns.*

```
columns = ['apple_pie?', 'potato_salad?', 'sushi?']
```

Let's wrap the final step in a function so we can reuse it as needed. It will have one parameter, the list of columns we want to find the highest information gain for.

```
def highest_info_gain(columns):
```

We'll intialize an empty dictionary to store our information gains.

```
information_gains = {}
```

And then we can iterate through the list of columns and store the result in our information_gains dictionary.

```
for col in columns:
    information_gain = calc_information_gain(midwest, col, 'midwest?)
    information_gains[col] = information_gain
```

Finally, we can return the key of the highest value in our dictionary.

```
return max(information_gains, key=information_gains.get)
```

All together now:

Once we execute our final function

```
print(highest_info_gain(midwest, columns, 'midwest?'))
//sushi
```

we see the variable/column with the highest information gain is ‘sushi?’.

We can visualize a split on sushi below:

Splitting our dataset on the sushi column

Our left split has two people out of six from the midwest. The right split has eight out of the nine people from the midwest. This was an efficient split and lowered our entropy on both sides. If we were to continue we would use recursion to keep splitting each split with a goal to end each branch with an entropy of zero.

Conclusion

Decision trees can be a useful machine learning algorithm to pick up nonlinear interactions between variables in the data. In this example, we looked at the beginning stages of a decision tree classification algorithm. We then looked at three information theory concepts, entropy, bit, and information gain. By using these concepts we were able to build a few functions in Python to decide which variables/columns were the most efficient to split on. With a firm grasp on these concepts, we can move forward to build a decision tree.

Data Science Coding Python Machine Learning Algorithms