**One-Hot-Encoding**

Consider following data. We’d like to write a decision tree to predict whether someone will like Troll 2.

|  |  |  |
| --- | --- | --- |
| **Favorite Color** | **Height (m)** | **Loves Troll 2** |
| Blue | 1.77 | Yes |
| Red | 1.32 | No |
| Green | 1.81 | Yes |
| Blue | 1.56 | No |
| Green | 1.64 | Yes |
| Green | 1.61 | No |
| Blue | 1.73 | No |

The methods we’ve used so far will work just fine. But some machine learning algorithms don’t like/can’t handle having columns with non-numerical data. So to use *them*, we’d have to convert Favorite Color to numerical values. One possibility is,

**Label-Encoding**

We could just say Red = 0, Blue = 1, Green = 2, for instance.

|  |  |  |
| --- | --- | --- |
| **Favorite Color** | **Height (m)** | **Loves Troll 2** |
| Blue = 1 | 1.77 | Yes |
| Red = 0 | 1.32 | No |
| Green = 2 | 1.81 | Yes |
| Blue = 1 | 1.56 | No |
| Green = 2 | 1.64 | Yes |
| Green = 2 | 1.61 | No |
| Blue = 1 | 1.73 | No |

But a problem with this is that, as we saw in the previous file, our decision tree algorithm would partition the Favorite Color data with possible FCcrit values of 0.5 and 1.5, and see which gave the lowest GI(LT2|FC) or S(LT2|FC) values. Thus, we’d separate colors into {(red), (blue, green)}, or {(red, blue), (green)} subsets. But if we had labelled Red = 0, Green = 1, Blue = 2, then we would’ve separated the colors into {(red), (green, blue)}, and {(red, green), (blue)} subsets. So it seems that it matterss what the numbers are, even though it shouldn’t. So this isn’t a great way to go about the conversion to numerical values.

On the other hand, if our categorical data were letter grades = A, B, C, D, F. Then it’d probably make sense to numerically encode them as 4, 3, 2, 1, 0.

**Target-Encoding**

One way to redress this is to replace the label, Ai, with P(Y|Ai).



which is the probability of a favorable outcome, given the value Ai (assuming Y has only two outcomes). So every label will be replaced by a ratio. I think the point of doing this is to group together labels with a prima facie similar causal relationship to Y (similar because they beget Y at similar rate). This coheres with the general supposition that *numerical* data that’s close together, should also, kind of by continuity arguments, have similar causal relationships vis a vis Y. So then we’d have:

|  |  |  |
| --- | --- | --- |
| **Favorite Color** | **Height (m)** | **Loves Troll 2** |
| Blue = 1/3 | 1.77 | Yes |
| Red = 0/1 | 1.32 | No |
| Green = 2/3 | 1.81 | Yes |
| Blue = 1/3 | 1.56 | No |
| Green = 2/3 | 1.64 | Yes |
| Green = 2/3 | 1.61 | No |
| Blue = 1/3 | 1.73 | No |

And now can see that when we do the aforementioned Decision Tree analysis on the Favorite Color data by devising some FCcrit cutoff, and grouping data < FCcrit together, and data > FCcrit together, etc., the data that has similar P(Y|Ai) will be grouped together. This is better than the arbitrary grouping from before.

But a drawback of this approach, is that it gives too much weight to data with few instances. For instance Favorite Color = Red has only one instance, and so how sure are we that Reds will in general have values ~ P(Y|A)? Well if Favorite Color were all the same value, say Blue, then the value we’d give it is P(Y|Ai) = P(Y) itself. So can think that the default value of any color should just be the overall P(Y). And then as we add more rows of a given color, we can take its value P(Y|Ai) more seriously. So we use a weighted average:



where m = number of rows of A = Ai data, and n is some overall weighting factor chosen by the user (n = 1 and n = total number of rows of data are both popular options). Note mP(Y = Yes|Ai) is just the number of Y = Yes instances for A = Ai. Let’s use n = number of rows of data altogether. Then we’d have:



and so,

|  |  |  |
| --- | --- | --- |
| **Favorite Color** | **Height (m)** | **Loves Troll 2** |
| Blue = 0.4 | 1.77 | Yes |
| Red = 0.37 | 1.32 | No |
| Green = 0.5 | 1.81 | Yes |
| Blue = 0.4 | 1.56 | No |
| Green = 0.5 | 1.64 | Yes |
| Green = 0.5 | 1.61 | No |
| Blue = 0.4 | 1.73 | No |

Not sure how this helps things, but. One problem with this approach is data leakage: we are incestuously using training data to modify our training data. And another problem with is that we would have to use these numerical labels, derived from the training data outcomes, to label any testing data. But then the testing data is dependent on the training data, not independent from it. And so that spoils the effectiveness of the testing data. One way to avoid this is **K-fold Target Encoding**. We start by splitting the data up into K equally sized (as near as can manage) groups. And then for each subgroup, we use only the data from all the other K-1 groups to encode our chosen subgroup’s numerical values. To illustrate. Let’s take K = 3, and split as follows,

|  |  |  |
| --- | --- | --- |
| **Favorite Color** | **Height (m)** | **Loves Troll 2** |
| Blue | 1.77 | Yes |
| Red | 1.32 | No |
| Green | 1.81 | Yes |
| Blue | 1.56 | No |
| Green | 1.64 | Yes |
| Green | 1.61 | No |
| Blue | 1.73 | No |

Then for our orange group, we use the blue + brown group’s values to encode. So,



Then for our blue group, we use the orange + brown group’s values to encode. So,



And last, for our brown group, we use the orange + blue group’s values to encode. So,



And now we fill these values in:

|  |  |  |
| --- | --- | --- |
| **Favorite Color** | **Height (m)** | **Loves Troll 2** |
| Blue = 0.17 | 1.77 | Yes |
| Red = 0.25 | 1.32 | No |
| Green = 0.33 | 1.81 | Yes |
| Blue = 0.42 | 1.56 | No |
| Green = 0.42 | 1.64 | Yes |
| Green = 0.57 | 1.61 | No |
| Blue = 0.72 | 1.73 | No |

There is a variant on Target Encoding called **Category Boost Target Encoding**. This is designed to mitigate the leakage problem. The formula we use to convert categories to numbers is similar to the one used above w/ n = 1, but instead of using all rows to calculate the numbers, we only use *previous* rows. Using only *previous* rows prevents a row from influencing itself – which is leakage. Formula is:



where m = # *previous* rows of that categorical value Ai, P(Y = Yes|Ai) is probability of Y given *previous* instances of Ai, and k is some user specified number. Note mP(Y = Yes|Ai) is just the number of previous Y = Yes instances for the categorical value A = Ai. And by *previous* row, we mean a row *above* the present one. So for instance, using this table,

|  |  |  |
| --- | --- | --- |
| **Favorite Color** | **Height (m)** | **Loves Troll 2** |
| Blue | 1.77 | Yes |
| Red | 1.32 | No |
| Green | 1.81 | Yes |
| Blue | 1.56 | No |
| Green | 1.64 | Yes |
| Green | 1.61 | No |
| Blue | 1.73 | No |

we’d do the following conversions, using, say k = 0.05:



And so we’d have:

|  |  |  |
| --- | --- | --- |
| **Favorite Color** | **Height (m)** | **Loves Troll 2** |
| Blue = 0.05 | 1.77 | Yes |
| Red = 0.05 | 1.32 | No |
| Green = 0.05 | 1.81 | Yes |
| Blue = 0.525 | 1.56 | No |
| Green = 0.525 | 1.64 | Yes |
| Green = 0.683 | 1.61 | No |
| Blue = 0.35 | 1.73 | No |

It is apparent that we’ll get a different encoding if we reorder the rows. Almost done.

**One-Hot Encoding**

Another approach is to just treat the different values as orthogonal vectors kind of. So if we have values Blue, Red, Green, then we assign to these a vector **FC** = (Blue, Red, Green). So we’d do,

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Blue** | **Red** | **Green** | **Height (m)** | **Loves Troll 2** |
| 1 | 0 | 0 | 1.77 | Yes |
| 0 | 1 | 0 | 1.32 | No |
| 0 | 0 | 1 | 1.81 | Yes |
| 1 | 0 | 0 | 1.56 | No |
| 0 | 0 | 1 | 1.64 | Yes |
| 0 | 0 | 1 | 1.61 | No |
| 1 | 0 | 0 | 1.73 | No |

This works well when we don’t have a whole lot of different values to work with (like just 3 colors). But if we have a lot, then might want to try one of the approaches above. Or could just use a ML Decision Tree Algorithm that allows non-numerical categorical data – have to look up what those are, but they’re out there. Note that before running this data through a ML model, you should drop one of the columns to avoid running into the so-called “colinearity” trap, which occurs when a column’s values are derivable from the other columns’ values. So we’d drop, say, Blue, column. Then we’d run the model on the other 3 independent columns/variables. To make a prediction for the first row, we’d feed in [0,0,1.77]. To make a prediction for the second row, we’d feed in [1,0,1.32]. And to make a predictioni for the third row, we’d feed in [0,1,1.81], etc. FWIW, I don’t think you have to explicitly drop any columns when using sklearn. I think it does this implicitly when making its analysis.

**Last Thing**

Oh yeah, and Loves Troll 2 values would all be converted to Yes = 1, No = 0 in all cases.