Smarter Ways to Encode Categorical Data for Machine Learning

Exploring Category Encoders

Jeff Hale Follow

Sep 10, 2018 · 12 min read

Better encoding of categorical data can mean better model performance. In this series I'll introduce you to a wide range of encoding options from the Category Encoders package for use with scikit-learn in Python.

Enigma for encoding

TL;DR;

Use Category Encoders to improve model performance when you have nominal or ordinal data that may provide value.

For nominal columns try OneHot, Hashing, LeaveOneOut, and Target encoding. Avoid OneHot for high cardinality columns and decision tree-based algorithms.

For ordinal columns try Ordinal (Integer), Binary, OneHot, LeaveOneOut, and Target. Helmert, Sum, BackwardDifference and Polynomial are less likely to be helpful, but if you have time or theoretic reason you might want to try them.

For regression tasks, Target and LeaveOneOut probably won't work well.

Roadmap

Map

In this article we'll discuss terms, general usage and five classic encoding options: Ordinal, One Hot, Binary, BaseN, and Hashing. In the future I may evaluate Bayesian encoders and contrast encoders with roots in statistical hypothesis testing.

In an earlier article I argued we should classify data as one of seven types to make better models faster. Here are the seven data types:

Useless—useless for machine learning algorithms, that is—discrete
Nominal—groups without order—discrete
Binary—either/or—discrete
Ordinal—groups with order—discrete
Count—the number of occurrences—discrete

Time—cyclical numbers with a temporal component—continuous
Interval—positive and/or negative numbers without a temporal component—continuous

Here we're concerned with encoding nominal and ordinal data. A column with nominal data has values that cannot be ordered in any meaningful way. Nominal data is most often one-hot (aka dummy) encoded, but there are many options that might perform better for machine learning.

Rank

In contrast, ordinal data can be rank ordered. Ordinal data can be encoded one of three ways, broadly speaking, but I think it's safe to say that its encoding is often not carefully considered.

1. It can be assumed to be close enough to interval data—with relatively equal magnitudes between the values—to treat it as such. Social scientists make this assumption all the time with Likert scales. For example, “On a scale from 1 to 7, 1 being extremely unlikely, 4 being neither likely nor unlikely and 7 being extremely likely, how likely are you to recommend this movie to a friend?”. Here the difference between 3 and 4 and the difference between 6 and 7 can be reasonably assumed to be similar.

2. It can be treated as nominal data, where each category has no numeric relationship to another. One-hot encoding and other encodings appropriate for nominal data make sense here.

3. The magnitude of the difference between the numbers can be ignored. You can just train your model with different encodings and seeing which encoding works best.

In this series we’ll look at Categorical Encoders 11 encoders as of version 1.2.8. **Update: Version 1.3.0 is the latest version on PyPI as of April 11, 2019.**

Many of these encoding methods go by more than one name in the statistics world and sometimes one name can mean different things. We’ll follow the Category Encoders usage.

Big thanks to Will McGinnis for creating and maintaining this package. It is largely derived from StatsModel’s Patsy package, which in turn is based on this UCLA statistics reference.

There are an infinite number of ways to encode categorical information. The ones in Category Encoders should be sufficient for most uses.

Quick Summary

Here's the list of Category Encoders functions with their descriptions and the type of data they would be most appropriate to encode.

Classic Encoders

The first group of five classic encoders can be seen on a continuum of embedding information in one column (Ordinal) up to *k* columns (OneHot). These are very useful encodings for machine learning practitioners to understand.

Ordinal—convert string labels to integer values 1 through *k*. Ordinal.
OneHot—one column for each value to compare vs. all other values. Nominal, ordinal.
Binary—convert each integer to binary digits. Each binary digit gets one column. Some info loss but fewer dimensions. Ordinal.
BaseN—Ordinal, Binary, or higher encoding. Nominal, ordinal. Doesn't add much functionality. Probably avoid.
Hashing—Like OneHot but fewer dimensions, some info loss due to collisions. Nominal, ordinal.

Contrast Encoders

The five contrast encoders all have multiple issues that I argue make them unlikely to be useful for machine learning. They all output one column for each column value. I would avoid them in most cases. Their stated intents are below.

***Helmert** (reverse)*—The mean of the dependent variable for a level is compared to the mean of the dependent variable over all previous levels.
Sum—compares the mean of the dependent variable for a given level to the overall mean of the dependent variable over all the levels.
Backward Difference—the mean of the dependent variable for a level is compared with the mean of the dependent variable for the prior level.
Polynomial—orthogonal polynomial contrasts. The coefficients taken on by polynomial coding for k=4 levels are the linear, quadratic, and cubic trends in the categorical variable.

Bayesian Encoders

The Bayesian encoders use information from the dependent variable in their encodings. They output one column and can work well with high cardinality data.

Target—use the mean of the DV, must take steps to avoid overfitting/ response leakage. Nominal, ordinal. For classification tasks.
LeaveOneOut—similar to target but avoids contamination. Nominal, ordinal. For classification tasks.
WeightOfEvidence—added in v1.3. Not documented in the docs as of April 11, 2019. The method is explained in this post.

James-Stein—forthcoming in v1.4. Described in the code here.
M-estimator—forthcoming in v1.4. Described in the code here.
Simplified target encoder.

Use

Category Encoders follow the same API as sklearn's preprocessors. They have some added conveniences, such as the ability to easily add an encoder to a pipeline. Additionally, the encoder returns a pandas DataFrame if a DataFrame is passed to it. Here's an example of the code with the BinaryEncoder:

```
# import the packages
import numpy as np
import pandas as pd
import category_encoders as ce

# make some data
df = pd.DataFrame({
 'color':["a", "b", "a", "c"],
 'outcome':[1, 2, 3, 2]})

# split into X and y
X = df.drop('outcome', axis = 1)
y = df.drop('color', axis = 1)
```

We'll tackle a few gotchas with implementation in the future. But you should be able to jump right into the first five if you are familiar with

scikit learn's api.

Note that all Category Encoders impute missing values automatically by default. However, I recommend filling missing data data yourself prior to encoding so you can test the results of several methods. I plan to discuss imputing options in a forthcoming article, so follow me on Medium if you want to make sure you don't miss it.

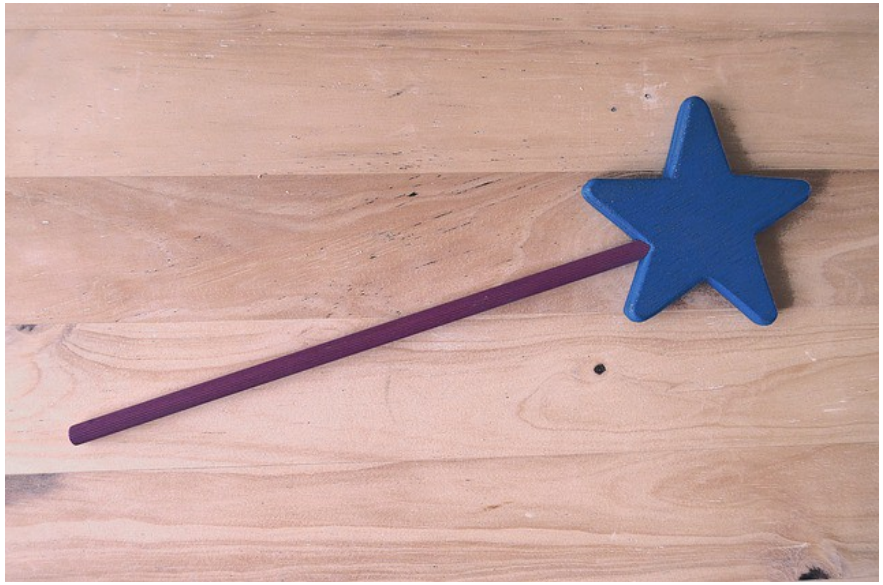
Terminology

You might see commentators use the following terms interchangeably: *dimension*, *feature*, *vector*, *series*, *independent variable*, and *column*. I will too :) Similarly, you might see *row* and *observation* used interchangeably.

k is the original number of unique values in your data column. *High cardinality* means a lot of unique values (a large *k)*. A column with hundreds of zip codes is an example of a high cardinality feature.

High cardinality theme bird

High dimensionality means a matrix with many dimensions. High dimensionality comes with the Curse of Dimensionality—a thorough treatment of this topic can be found here. The take away is that high dimensionality requires many observations and often results in overfitting.



A wand to help ward off the Curse of Dimensionality

Sparse data is a matrix with lots of zeroes relative to other values. If your encoders transform your data so that it becomes sparse, some algorithms may not work well. Sparsity can often be managed by flagging it, but many algorithms don't work well unless the data is dense.

Sparse

Digging Into Category Encoders

Without further ado, let's encode!

Ordinal

OrdinalEncoder converts each string value to a whole number. The first unique value in your column becomes 1, the second becomes 2, the third becomes 3, and so on.

What the actual value was prior to encoding does not affect what it becomes when you *fit_transform* with OrdinalEncoder. The first value

could have been 10 and the second value could have been 3. Now they will be 1 and 3, respectively.

If the column contains nominal data, stopping after you use OrdinalEncoder is a bad idea. Your machine learning algorithm will treat the variable as continuous and assume the values are on a meaningful scale. Instead, if you have a column with values *car, bus,* and *truck* you should first encode this nominal data using OrdinalEncoder. Then encode it again using one of the methods appropriate to nominal data that we'll explore below.

In contrast, if your column values are truly ordinal, that means that the integer assigned to each value is meaningful. Assignment should be done with intention. Say your column had the string values "First", "Third", and "Second" in it. Those values should be mapped to the corresponding integers by passing OrdinalEncoder a list of dicts like so:

```
[{'col': 'finished_race_order',
  'mapping': [("First", 1),
              ('Second', 2),
              ('Third', 3)]
}]
```

Here's the basic setup for all the code samples to follow. You can get the full notebook at this Kaggle Kernel.

```
import numpy as np
import pandas as pd              # version 0.23.4
import category_encoders as ce   # version 1.2.8
from sklearn.preprocessing import LabelEncoder

pd.options.display.float_format = '{:.2f}'.format # to make legible

# make some data
df = pd.DataFrame({
    'color':["a", "c", "a", "a", "b", "b"],
    'outcome':[1, 2, 0, 0, 0, 1]})

# set up X and y
X = df.drop('outcome', axis = 1)
y = df.drop('color', axis = 1)
```

Here's the untransformed X column.

```
X
```

	color
0	a
1	c
2	a
3	a
4	b
5	b

And here's the OrdinalEncoder code to transform the *color* column values from letters to integers.

```
ce_ord = ce.OrdinalEncoder(cols = ['color'])
ce_ord.fit_transform(X, y['outcome'])
```

	color
0	1
1	2
2	1
3	1
4	3
5	3

All the string values are now integers.

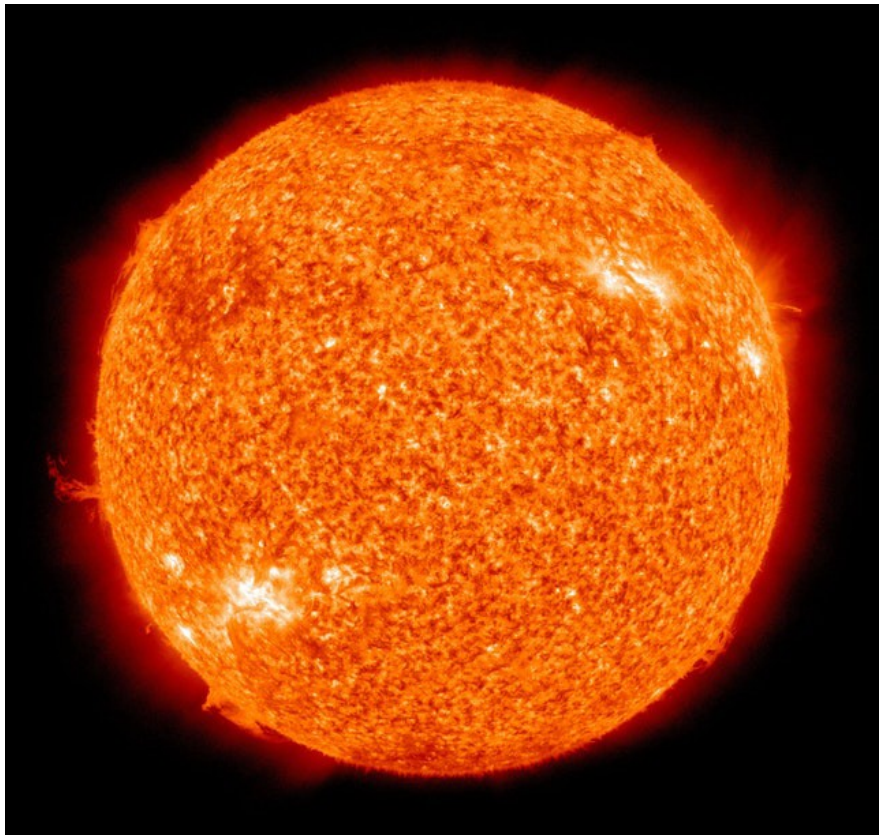
Sklearn's LabelEncoder does pretty much the same thing as Category Encoder's OrdinalEncoder, but is not quite as user friendly. LabelEncoder won't return a DataFrame, instead it returns a numpy array if you pass a DataFrame. It also outputs values starting with 0, compared to OrdinalEncoder's default of outputting values starting with 1.

You could accomplish ordinal encoding by mapping string values to integers manually with *apply*. But that's extra work once you know how to use Category Encoders.

OneHot

One-hot encoding is the classic approach to dealing with nominal, and maybe ordinal, data. It's referred to as the "The Standard Approach for Categorical Data" in Kaggle's Machine Learning tutorial series. It also

goes by the names *dummy* encoding, *indicator* encoding, and occasionally *binary* encoding. Yes, this is confusing.



That's one hot sun

The one-hot encoder creates one column for each value to compare against all other values. For each new column, a row gets a 1 if the row contained that column's value and a 0 if it did not. Here's how it looks:

```
ce_one_hot = ce.OneHotEncoder(cols = ['color'])
ce_one_hot.fit_transform(X, y)
```

	color_1	color_2	color_3	color_-1
0	1	0	0	0
1	0	1	0	0
2	1	0	0	0
3	1	0	0	0
4	0	0	1	0
5	0	0	1	0

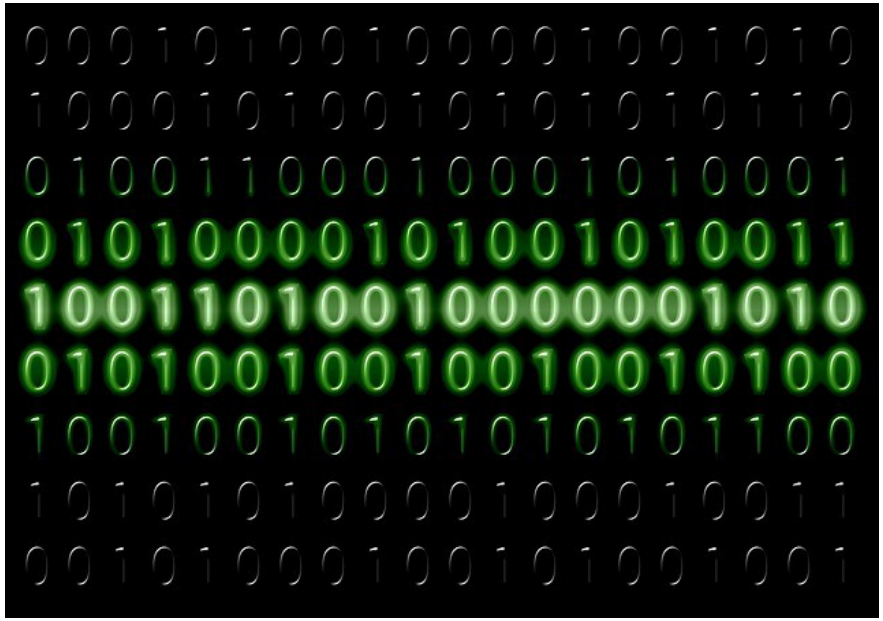
color_-1 is actually an extraneous column, because it's all 0s—with no variation it's not helping your model learn anything. It may have been intended for missing values, but in version 1.2.8 of Category Encoders it isn't doing anything. However, it's only adding one column so it's not really a big deal for performance.

One-hot encoding can perform very well, but the number of new features is equal to *k,* the number of unique values. This feature expansion can create serious memory problems if your data set has high cardinality features. One-hot-encoded data can also be difficult for decision-tree-based algorithms—see discussion here.

The pandas GetDummies and sklearn OneHotEncoder functions perform the same role as OneHotEncoder. I find OneHotEncoder a bit nicer to use.

Binary

Binary can be thought of as a hybrid of one-hot and hashing encoders. Binary creates fewer features than one-hot, while preserving some uniqueness of values in the the column. It can work well with higher dimensionality ordinal data.



Binary

Here's how it works:

- The categories are encoded by OrdinalEncoder if they aren't already in numeric form.
- Then those integers are converted into binary code, so for example 5 becomes 101 and 10 becomes 1010

- Then the digits from that binary string are split into separate columns. So if there are 4–7 values in an ordinal column then 3 new columns are created: one for the first bit, one for the second, and one for the third.
- Each observation is encoded across the columns in its binary form.

Here's how it looks:

```
ce_bin = ce.BinaryEncoder(cols = ['color'])
ce_bin.fit_transform(X, y)
```

	color_0	color_1	color_2
0	0	0	1
1	0	1	0
2	0	0	1
3	0	0	1
4	0	1	1
5	0	1	1

The first column has no variance, so it isn't doing anything to help the model.

With only three levels, the information embedded becomes muddled. There are many collisions and the model can't glean much information from the features. Just one-hot encode a column if it only has a few values.

In contrast, binary really shines when the cardinality of the column is higher—with the 50 US states, for example.

Binary encoding creates fewer columns than one-hot encoding. It is more memory efficient. It also reduces the chances of dimensionality problems with higher cardinality.

Most similar values overlap with each other across many of the new columns. This allows many machine learning algorithms to learn the values similarity. Binary encoding is a decent compromise for ordinal data with high cardinality.

For nominal data a hashing algorithm with more fine-grained control usually makes more sense. If you've used binary encoding successfully, please share in the comments.

BaseN

When the BaseN *base = 1* it is basically the same as one hot encoding. When *base = 2* it is basically the same as binary encoding. McGinnis said, "Practically, this adds very little new functionality, rarely do people use base-3 or base-8 or any base other than ordinal or binary in real problems."

Base 3

The main reason for its existence is to possibly make grid searching easier. You could use BaseN with *gridsearchCV*. However, if you're going to grid search with some of these encoding options, you're going to make that search part of your workflow anyway. I don't see a compelling reason to use BaseN. If you do, please share in the comments.

```
ce_basen = ce.BaseNEncoder(cols = ['color'])
ce_basen.fit_transform(X, y)
```

	color_0	color_1	color_2
0	0	0	1
1	0	1	0
2	0	0	1
3	0	0	1
4	0	1	1
5	0	1	1

The default base for BaseNEncoder is 2, which is the equivalent of BinaryEncoder.

Hashing

HashingEncoder implements the hashing trick. It is similar to one-hot encoding but with fewer new dimensions and some info loss due to collisions. The collisions do not significantly affect performance unless there is a great deal of overlap. An excellent discussion of the hashing trick and guidelines for selecting the number of output features can be found here.

Here's the ordinal column again for a refresher.

```
1 X
```

	color
0	a
1	c
2	a
3	a
4	b
5	b

And here's the HashingEncoder.

```
1 ce_hash = ce.HashingEncoder(cols = ['color'])
2 ce_hash.fit_transform(X, y)
```

	col_0	col_1	col_2	col_3	col_4	col_5	col_6	col_7
0	0	1	0	0	0	0	0	0
1	0	0	0	1	0	0	0	0
2	0	1	0	0	0	0	0	0
3	0	1	0	0	0	0	0	0
4	0	0	0	0	0	0	0	1
5	0	0	0	0	0	0	0	1

The *n_components* parameter controls the number of expanded columns. The default is eight columns. In our example column with three values the default results in five columns full of 0s.

If you set *n_components* less than *k* you'll have a small reduction in the value provided by the encoded data. You'll also have fewer dimensions.

You can pass a hashing algorithm of your choice to HashingEncoder; the default is *md5*. Hashing algorithms have been very successful in some Kaggle competitions. It's worth trying HashingEncoder for nominal and ordinal data if you have high cardinality features.

Wrap

Exercise break

That's all for now. Here's a recap and suggestions for remaining encoders.

For nominal columns try OneHot, Hashing, LeaveOneOut, and Target encoding. Avoid OneHot for high cardinality columns and decision tree-based algorithms.

For ordinal columns try Ordinal (Integer), Binary, OneHot, LeaveOneOut, and Target. Helmert, Sum, BackwardDifference and Polynomial are less likely to be helpful, but if you have time or theoretic reason you might want to try them.

The Bayesian encoders can work well for some machine learning tasks. For example, Owen Zhang used the leave one out encoding method to perform well in a Kaggle classification challenge.

*Update April 2019: I updated this article to include information about forthcoming encoders and reworked the conclusion.**

I write about data science, Python, and DevOps. Check out my other articles and follow me here, if you're into that stuff.

If you found this article interesting, helpful, or mildly amusing, please help others find it with a few clappity claps. 😀

Thanks for reading!