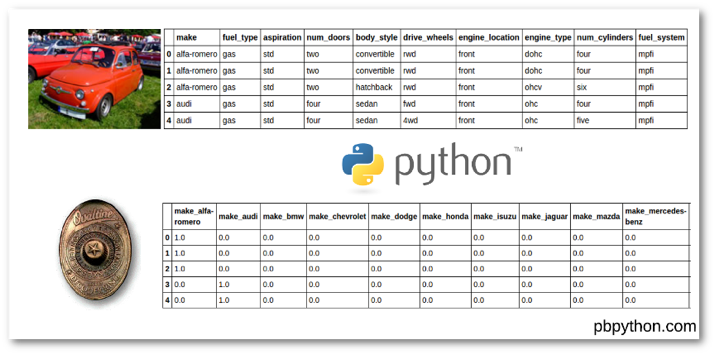
Mon 06 February 2017

[Guide to Encoding Categorical Values in Python](https://pbpython.com/categorical-encoding.html)

Posted by [Chris Moffitt](https://pbpython.com/author/chris-moffitt.html) in [articles](https://pbpython.com/category/articles.html)



Introduction

In many practical Data Science activities, the data set will contain categorical variables. These variables are typically stored as text values which represent various traits. Some examples include color (“Red”, “Yellow”, “Blue”), size (“Small”, “Medium”, “Large”) or geographic designations (State or Country). Regardless of what the value is used for, the challenge is determining how to use this data in the analysis. Many machine learning algorithms can support categorical values without further manipulation but there are many more algorithms that do not. Therefore, the analyst is faced with the challenge of figuring out how to turn these text attributes into numerical values for further processing.

As with many other aspects of the Data Science world, there is no single answer on how to approach this problem. Each approach has trade-offs and has potential impact on the outcome of the analysis. Fortunately, the python tools of pandas and scikit-learn provide several approaches that can be applied to transform the categorical data into suitable numeric values. This article will be a survey of some of the various common (and a few more complex) approaches in the hope that it will help others apply these techniques to their real world problems.

The Data Set

For this article, I was able to find a good dataset at the [UCI Machine Learning Repository](https://archive.ics.uci.edu/ml/index.php). This particular [Automobile Data Set](https://archive.ics.uci.edu/ml/datasets/automobile) includes a good mix of categorical values as well as continuous values and serves as a useful example that is relatively easy to understand. Since domain understanding is an important aspect when deciding how to encode various categorical values - this data set makes a good case study.

Before we get started encoding the various values, we need to important the data and do some minor cleanups. Fortunately, pandas makes this straightforward:

**import** **pandas** **as** **pd**

**import** **numpy** **as** **np**

*# Define the headers since the data does not have any*

headers = ["symboling", "normalized\_losses", "make", "fuel\_type", "aspiration",

"num\_doors", "body\_style", "drive\_wheels", "engine\_location",

"wheel\_base", "length", "width", "height", "curb\_weight",

"engine\_type", "num\_cylinders", "engine\_size", "fuel\_system",

"bore", "stroke", "compression\_ratio", "horsepower", "peak\_rpm",

"city\_mpg", "highway\_mpg", "price"]

*# Read in the CSV file and convert "?" to NaN*

df = pd.read\_csv("https://archive.ics.uci.edu/ml/machine-learning-databases/autos/imports-85.data",

header=**None**, names=headers, na\_values="?" )

df.head()

|  | **symboling** | **normalized\_losses** | **make** | **fuel\_type** | **aspiration** | **num\_doors** | **body\_style** | **drive\_wheels** | **engine\_location** | **wheel\_base** | **…** | **engine\_size** | **fuel\_system** | **bore** | **stroke** | **compression\_ratio** | **horsepower** | **peak\_rpm** | **city\_mpg** | **highway\_mpg** | **price** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | 3 | NaN | alfa-romero | gas | std | two | convertible | rwd | front | 88.6 | … | 130 | mpfi | 3.47 | 2.68 | 9.0 | 111.0 | 5000.0 | 21 | 27 | 13495.0 |
| **1** | 3 | NaN | alfa-romero | gas | std | two | convertible | rwd | front | 88.6 | … | 130 | mpfi | 3.47 | 2.68 | 9.0 | 111.0 | 5000.0 | 21 | 27 | 16500.0 |
| **2** | 1 | NaN | alfa-romero | gas | std | two | hatchback | rwd | front | 94.5 | … | 152 | mpfi | 2.68 | 3.47 | 9.0 | 154.0 | 5000.0 | 19 | 26 | 16500.0 |
| **3** | 2 | 164.0 | audi | gas | std | four | sedan | fwd | front | 99.8 | … | 109 | mpfi | 3.19 | 3.40 | 10.0 | 102.0 | 5500.0 | 24 | 30 | 13950.0 |
| **4** | 2 | 164.0 | audi | gas | std | four | sedan | 4wd | front | 99.4 | … | 136 | mpfi | 3.19 | 3.40 | 8.0 | 115.0 | 5500.0 | 18 | 22 | 17450.0 |

The final check we want to do is see what data types we have:

df.dtypes

symboling int64

normalized\_losses float64

make object

fuel\_type object

aspiration object

num\_doors object

body\_style object

drive\_wheels object

engine\_location object

wheel\_base float64

length float64

width float64

height float64

curb\_weight int64

engine\_type object

num\_cylinders object

engine\_size int64

fuel\_system object

bore float64

stroke float64

compression\_ratio float64

horsepower float64

peak\_rpm float64

city\_mpg int64

highway\_mpg int64

price float64

dtype: object

Since this article will only focus on encoding the categorical variables, we are going to include only the object columns in our dataframe. Pandas has a helpful select\_dtypes function which we can use to build a new dataframe containing only the object columns.

obj\_df = df.select\_dtypes(include=['object']).copy()

obj\_df.head()

|  | **make** | **fuel\_type** | **aspiration** | **num\_doors** | **body\_style** | **drive\_wheels** | **engine\_location** | **engine\_type** | **num\_cylinders** | **fuel\_system** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | alfa-romero | gas | std | two | convertible | rwd | front | dohc | four | mpfi |
| **1** | alfa-romero | gas | std | two | convertible | rwd | front | dohc | four | mpfi |
| **2** | alfa-romero | gas | std | two | hatchback | rwd | front | ohcv | six | mpfi |
| **3** | audi | gas | std | four | sedan | fwd | front | ohc | four | mpfi |
| **4** | audi | gas | std | four | sedan | 4wd | front | ohc | five | mpfi |

Before going any further, there are a couple of null values in the data that we need to clean up.

obj\_df[obj\_df.isnull().any(axis=1)]

|  | **make** | **fuel\_type** | **aspiration** | **num\_doors** | **body\_style** | **drive\_wheels** | **engine\_location** | **engine\_type** | **num\_cylinders** | **fuel\_system** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **27** | dodge | gas | turbo | NaN | sedan | fwd | front | ohc | four | mpfi |
| **63** | mazda | diesel | std | NaN | sedan | fwd | front | ohc | four | idi |

For the sake of simplicity, just fill in the value with the number 4 (since that is the most common value):

obj\_df["num\_doors"].value\_counts()

four 114

two 89

Name: num\_doors, dtype: int64

obj\_df = obj\_df.fillna({"num\_doors": "four"})

Now that the data does not have any null values, we can look at options for encoding the categorical values.

Approach #1 - Find and Replace

Before we go into some of the more “standard” approaches for encoding categorical data, this data set highlights one potential approach I’m calling “find and replace.”

There are two columns of data where the values are words used to represent numbers. Specifically the number of cylinders in the engine and number of doors on the car. Pandas makes it easy for us to directly replace the text values with their numeric equivalent by using replace .

We have already seen that the num\_doors data only includes 2 or 4 doors. The number of cylinders only includes 7 values and they are easily translated to valid numbers:

obj\_df["num\_cylinders"].value\_counts()

four 159

six 24

five 11

eight 5

two 4

twelve 1

three 1

Name: num\_cylinders, dtype: int64

If you review the replace [documentation](http://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.replace.html), you can see that it is a powerful command that has many options. For our uses, we are going to create a mapping dictionary that contains each column to process as well as a dictionary of the values to translate.

Here is the complete dictionary for cleaning up the num\_doors and num\_cylinders columns:

cleanup\_nums = {"num\_doors": {"four": 4, "two": 2},

"num\_cylinders": {"four": 4, "six": 6, "five": 5, "eight": 8,

"two": 2, "twelve": 12, "three":3 }}

To convert the columns to numbers using replace :

obj\_df = obj\_df.replace(cleanup\_nums)

obj\_df.head()

|  | **make** | **fuel\_type** | **aspiration** | **num\_doors** | **body\_style** | **drive\_wheels** | **engine\_location** | **engine\_type** | **num\_cylinders** | **fuel\_system** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | alfa-romero | gas | std | 2 | convertible | rwd | front | dohc | 4 | mpfi |
| **1** | alfa-romero | gas | std | 2 | convertible | rwd | front | dohc | 4 | mpfi |
| **2** | alfa-romero | gas | std | 2 | hatchback | rwd | front | ohcv | 6 | mpfi |
| **3** | audi | gas | std | 4 | sedan | fwd | front | ohc | 4 | mpfi |
| **4** | audi | gas | std | 4 | sedan | 4wd | front | ohc | 5 | mpfi |

The nice benefit to this approach is that pandas “knows” the types of values in the columns so the object is now a int64

obj\_df.dtypes

make object

fuel\_type object

aspiration object

num\_doors int64

body\_style object

drive\_wheels object

engine\_location object

engine\_type object

num\_cylinders int64

fuel\_system object

dtype: object

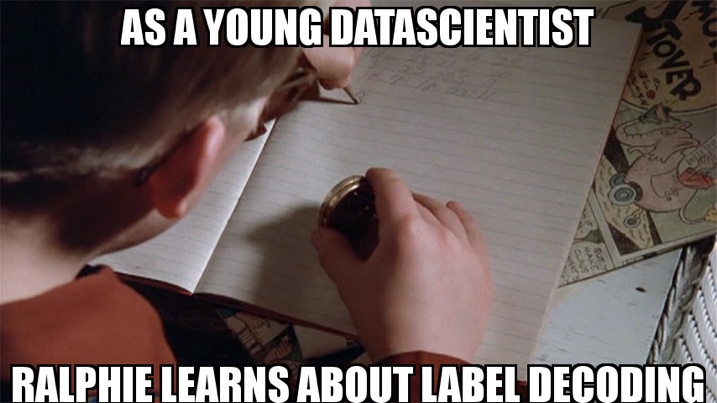
While this approach may only work in certain scenarios it is a very useful demonstration of how to convert text values to numeric when there is an “easy” human interpretation of the data. This concept is also useful for more general data cleanup.

Approach #2 - Label Encoding

Another approach to encoding categorical values is to use a technique called label encoding. Label encoding is simply converting each value in a column to a number. For example, the body\_style column contains 5 different values. We could choose to encode it like this:

* convertible -> 0
* hardtop -> 1
* hatchback -> 2
* sedan -> 3
* wagon -> 4

This process reminds me of Ralphie using his secret decoder ring in “A Christmas Story”



One trick you can use in pandas is to convert a column to a category, then use those category values for your label encoding:

obj\_df["body\_style"] = obj\_df["body\_style"].astype('category')

obj\_df.dtypes

make object

fuel\_type object

aspiration object

num\_doors int64

body\_style category

drive\_wheels object

engine\_location object

engine\_type object

num\_cylinders int64

fuel\_system object

dtype: object

Then you can assign the encoded variable to a new column using the cat.codes accessor:

obj\_df["body\_style\_cat"] = obj\_df["body\_style"].cat.codes

obj\_df.head()

|  | **make** | **fuel\_type** | **aspiration** | **num\_doors** | **body\_style** | **drive\_wheels** | **engine\_location** | **engine\_type** | **num\_cylinders** | **fuel\_system** | **body\_style\_cat** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | alfa-romero | gas | std | 2 | convertible | rwd | front | dohc | 4 | mpfi | 0 |
| **1** | alfa-romero | gas | std | 2 | convertible | rwd | front | dohc | 4 | mpfi | 0 |
| **2** | alfa-romero | gas | std | 2 | hatchback | rwd | front | ohcv | 6 | mpfi | 2 |
| **3** | audi | gas | std | 4 | sedan | fwd | front | ohc | 4 | mpfi | 3 |
| **4** | audi | gas | std | 4 | sedan | 4wd | front | ohc | 5 | mpfi | 3 |

The nice aspect of this approach is that you get the benefits of pandas categories (compact data size, ability to order, plotting support) but can easily be converted to numeric values for further analysis.

Approach #3 - One Hot Encoding

Label encoding has the advantage that it is straightforward but it has the disadvantage that the numeric values can be “misinterpreted” by the algorithms. For example, the value of 0 is obviously less than the value of 4 but does that really correspond to the data set in real life? Does a wagon have “4X” more weight in our calculation than the convertible? In this example, I don’t think so.

A common alternative approach is called one hot encoding (but also goes by several different names shown below). Despite the different names, the basic strategy is to convert each category value into a new column and assigns a 1 or 0 (True/False) value to the column. This has the benefit of not weighting a value improperly but does have the downside of adding more columns to the data set.

Pandas supports this feature using [get\_dummies](http://pandas.pydata.org/pandas-docs/stable/generated/pandas.get_dummies.html). This function is named this way because it creates dummy/indicator variables (aka 1 or 0).

Hopefully a simple example will make this more clear. We can look at the column drive\_wheels where we have values of 4wd , fwd or rwd . By using get\_dummies we can convert this to three columns with a 1 or 0 corresponding to the correct value:

pd.get\_dummies(obj\_df, columns=["drive\_wheels"]).head()

|  | **make** | **fuel\_type** | **aspiration** | **num\_doors** | **body\_style** | **engine\_location** | **engine\_type** | **num\_cylinders** | **fuel\_system** | **body\_style\_cat** | **drive\_wheels\_4wd** | **drive\_wheels\_fwd** | **drive\_wheels\_rwd** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | alfa-romero | gas | std | 2 | convertible | front | dohc | 4 | mpfi | 0 | 0.0 | 0.0 | 1.0 |
| **1** | alfa-romero | gas | std | 2 | convertible | front | dohc | 4 | mpfi | 0 | 0.0 | 0.0 | 1.0 |
| **2** | alfa-romero | gas | std | 2 | hatchback | front | ohcv | 6 | mpfi | 2 | 0.0 | 0.0 | 1.0 |
| **3** | audi | gas | std | 4 | sedan | front | ohc | 4 | mpfi | 3 | 0.0 | 1.0 | 0.0 |
| **4** | audi | gas | std | 4 | sedan | front | ohc | 5 | mpfi | 3 | 1.0 | 0.0 | 0.0 |

The new data set contains three new columns:

* drive\_wheels\_4wd
* drive\_wheels\_rwd
* drive\_wheels\_fwd

This function is powerful because you can pass as many category columns as you would like and choose how to label the columns using prefix . Proper naming will make the rest of the analysis just a little bit easier.

pd.get\_dummies(obj\_df, columns=["body\_style", "drive\_wheels"], prefix=["body", "drive"]).head()

|  | **make** | **fuel\_type** | **aspiration** | **num\_doors** | **engine\_location** | **engine\_type** | **num\_cylinders** | **fuel\_system** | **body\_style\_cat** | **body\_convertible** | **body\_hardtop** | **body\_hatchback** | **body\_sedan** | **body\_wagon** | **drive\_4wd** | **drive\_fwd** | **drive\_rwd** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | alfa-romero | gas | std | 2 | front | dohc | 4 | mpfi | 0 | 1.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 1.0 |
| **1** | alfa-romero | gas | std | 2 | front | dohc | 4 | mpfi | 0 | 1.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 1.0 |
| **2** | alfa-romero | gas | std | 2 | front | ohcv | 6 | mpfi | 2 | 0.0 | 0.0 | 1.0 | 0.0 | 0.0 | 0.0 | 0.0 | 1.0 |
| **3** | audi | gas | std | 4 | front | ohc | 4 | mpfi | 3 | 0.0 | 0.0 | 0.0 | 1.0 | 0.0 | 0.0 | 1.0 | 0.0 |
| **4** | audi | gas | std | 4 | front | ohc | 5 | mpfi | 3 | 0.0 | 0.0 | 0.0 | 1.0 | 0.0 | 1.0 | 0.0 | 0.0 |

The other concept to keep in mind is that get\_dummies returns the full dataframe so you will need to filter out the objects using select\_dtypes when you are ready to do the final analysis.

One hot encoding, is very useful but it can cause the number of columns to expand greatly if you have very many unique values in a column. For the number of values in this example, it is not a problem. However you can see how this gets really challenging to manage when you have many more options.

Approach #4 - Custom Binary Encoding

Depending on the data set, you may be able to use some combination of label encoding and one hot encoding to create a binary column that meets your needs for further analysis.

In this particular data set, there is a column called engine\_type that contains several different values:

obj\_df["engine\_type"].value\_counts()

ohc 148

ohcf 15

ohcv 13

l 12

dohc 12

rotor 4

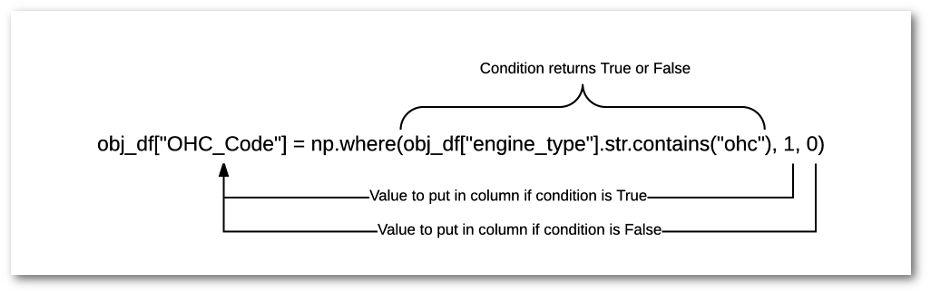
dohcv 1

Name: engine\_type, dtype: int64

For the sake of discussion, maybe all we care about is whether or not the engine is an Overhead Cam (OHC) or not. In other words, the various versions of OHC are all the same for this analysis. If this is the case, then we could use the str accessor plus np.where to create a new column the indicates whether or not the car has an OHC engine.

obj\_df["OHC\_Code"] = np.where(obj\_df["engine\_type"].str.contains("ohc"), 1, 0)

I find that this is a handy function I use quite a bit but sometimes forget the syntax so here is a graphic showing what we are doing:



The resulting dataframe looks like this (only showing a subset of columns):

obj\_df[["make", "engine\_type", "OHC\_Code"]].head()

|  | **make** | **engine\_type** | **OHC\_Code** |
| --- | --- | --- | --- |
| **0** | alfa-romero | dohc | 1 |
| **1** | alfa-romero | dohc | 1 |
| **2** | alfa-romero | ohcv | 1 |
| **3** | audi | ohc | 1 |
| **4** | audi | ohc | 1 |

This approach can be really useful if there is an option to consolidate to a simple Y/N value in a column. This also highlights how important domain knowledge is to solving the problem in the most efficient manner possible.

Scikit-Learn

scikit-learn updates

The previous version of this article used LabelEncoder and LabelBinarizer which are not the recommended approach for encoding categorical values. These encoders should only be used to encode the target values not the feature values.

The examples below use OrdinalEncoder and OneHotEncoder which is the correct approach to use for encoding target values.

In addition to the pandas approach, scikit-learn provides [similar functionality](http://scikit-learn.org/stable/modules/preprocessing.html#encoding-categorical-features). Personally, I find using pandas a little simpler to understand but the scikit approach is optimal when you are trying to build a predictive model.

For instance, if we want to do the equivalent to label encoding on the make of the car, we need to instantiate a OrdinalEncoder object and fit\_transform the data:

**from** **sklearn.preprocessing** **import** OrdinalEncoder

ord\_enc = OrdinalEncoder()

obj\_df["make\_code"] = ord\_enc.fit\_transform(obj\_df[["make"]])

obj\_df[["make", "make\_code"]].head(11)

|  | **make** | **make\_code** |
| --- | --- | --- |
| **0** | alfa-romero | 0 |
| **1** | alfa-romero | 0 |
| **2** | alfa-romero | 0 |
| **3** | audi | 1 |
| **4** | audi | 1 |
| **5** | audi | 1 |
| **6** | audi | 1 |
| **7** | audi | 1 |
| **8** | audi | 1 |
| **9** | audi | 1 |
| **10** | bmw | 2 |

Scikit-learn also supports binary encoding by using the OneHotEncoder. We use a similar process as above to transform the data but the process of creating a pandas DataFrame adds a couple of extra steps.

**from** **sklearn.preprocessing** **import** OneHotEncoder

oe\_style = OneHotEncoder()

oe\_results = oe\_style.fit\_transform(obj\_df[["body\_style"]])

pd.DataFrame(oe\_results.toarray(), columns=oe\_style.categories\_).head()

|  | **convertible** | **hardtop** | **hatchback** | **sedan** | **wagon** |
| --- | --- | --- | --- | --- | --- |
| **0** | 1 | 0 | 0 | 0 | 0 |
| **1** | 1 | 0 | 0 | 0 | 0 |
| **2** | 0 | 0 | 1 | 0 | 0 |
| **3** | 0 | 0 | 0 | 1 | 0 |
| **4** | 0 | 0 | 0 | 1 | 0 |

The next step would be to join this data back to the original dataframe. Here is an example:

obj\_df = obj\_df.join(pd.DataFrame(oe\_results.toarray(), columns=oe\_style.categories\_))

The key point is that you need to use toarray() to convert the results to a format that can be converted into a DataFrame.

Advanced Approaches

There are even more advanced algorithms for categorical encoding. I do not have a lot of personal experience with them but for the sake of rounding out this guide, I wanted to included them. This [article](http://www.willmcginnis.com/2015/11/29/beyond-one-hot-an-exploration-of-categorical-variables/) provides some additional technical background. The other nice aspect is that the author of the article has created a scikit-learn contrib package called [category\_encoders](https://github.com/scikit-learn-contrib/category_encoders) which implements many of these approaches. It is a very nice tool for approaching this problem from a different perspective.

Here is a brief introduction to using the library for some other types of encoding. For the first example, we will try doing a [Backward Difference encoding](https://stats.idre.ucla.edu/r/library/r-library-contrast-coding-systems-for-categorical-variables/#backward).

First we get a clean dataframe and setup the BackwardDifferenceEncoder :

**import** **category\_encoders** **as** **ce**

*# Get a new clean dataframe*

obj\_df = df.select\_dtypes(include=['object']).copy()

*# Specify the columns to encode then fit and transform*

encoder = ce.BackwardDifferenceEncoder(cols=["engine\_type"])

encoder.fit\_transform(obj\_df, verbose=1).iloc[:,8:14].head()

|  | **engine\_type\_0** | **engine\_type\_1** | **engine\_type\_2** | **engine\_type\_3** | **engine\_type\_4** | **engine\_type\_5** |
| --- | --- | --- | --- | --- | --- | --- |
| **0** | -0.857143 | -0.714286 | -0.571429 | -0.428571 | -0.285714 | -0.142857 |
| **1** | -0.857143 | -0.714286 | -0.571429 | -0.428571 | -0.285714 | -0.142857 |
| **2** | 0.142857 | -0.714286 | -0.571429 | -0.428571 | -0.285714 | -0.142857 |
| **3** | 0.142857 | 0.285714 | -0.571429 | -0.428571 | -0.285714 | -0.142857 |
| **4** | 0.142857 | 0.285714 | -0.571429 | -0.428571 | -0.285714 | -0.142857 |

The interesting thing is that you can see that the result are not the standard 1’s and 0’s we saw in the earlier encoding examples.

If we try a [polynomial encoding](https://stats.idre.ucla.edu/r/library/r-library-contrast-coding-systems-for-categorical-variables/#ORTHOGONAL), we get a different distribution of values used to encode the columns:

encoder = ce.PolynomialEncoder(cols=["engine\_type"])

encoder.fit\_transform(obj\_df, verbose=1).iloc[:,8:14].head()

|  | **engine\_type\_0** | **engine\_type\_1** | **engine\_type\_2** | **engine\_type\_3** | **engine\_type\_4** | **engine\_type\_5** |
| --- | --- | --- | --- | --- | --- | --- |
| **0** | -0.566947 | 0.545545 | -0.408248 | 0.241747 | -0.109109 | 0.032898 |
| **1** | -0.566947 | 0.545545 | -0.408248 | 0.241747 | -0.109109 | 0.032898 |
| **2** | -0.377964 | 0.000000 | 0.408248 | -0.564076 | 0.436436 | -0.197386 |
| **3** | -0.188982 | -0.327327 | 0.408248 | 0.080582 | -0.545545 | 0.493464 |
| **4** | -0.188982 | -0.327327 | 0.408248 | 0.080582 | -0.545545 | 0.493464 |

There are several different algorithms included in this package and the best way to learn is to try them out and see if it helps you with the accuracy of your analysis. The code shown above should give you guidance on how to plug in the other approaches and see what kind of results you get.

scikit-learn pipelines

Using pipelines

This section was added in November 2020. The goal is to show how to integrate the scikit-learn feature encoding functions into a simple model building pipeline.

As mentioned above, scikit-learn’s categorical encoders allow you to incorporate the transformation into your pipelines which can simplify the model building process and avoid some pitfalls. I recommend this Data School [video](https://www.dataschool.io/encoding-categorical-features-in-python/) as a good intro. It also serves as the basis for the approach outlined below.

Here is a very quick example of how to incorporate the OneHotEncoder and OrdinalEncoder into a pipeline and use cross\_val\_score to analyze the results:

**from** **sklearn.compose** **import** make\_column\_transformer

**from** **sklearn.linear\_model** **import** LinearRegression

**from** **sklearn.pipeline** **import** make\_pipeline

**from** **sklearn.model\_selection** **import** cross\_val\_score

*# for the purposes of this analysis, only use a small subset of features*

feature\_cols = [

'fuel\_type', 'make', 'aspiration', 'highway\_mpg', 'city\_mpg',

'curb\_weight', 'drive\_wheels'

]

*# Remove the empty price rows*

df\_ml = df.dropna(subset=['price'])

X = df\_ml[feature\_cols]

y = df\_ml['price']

Now that we have our data, let’s build the column transformer:

column\_trans = make\_column\_transformer((OneHotEncoder(handle\_unknown='ignore'),

['fuel\_type', 'make', 'drive\_wheels']),

(OrdinalEncoder(), ['aspiration']),

remainder='passthrough')

This example shows how to apply different encoder types for certain columns. Using the remainder='passthrough' argument to pass all the numeric values through the pipeline without any changes.

For the model, we use a simple linear regression and then make the pipeline:

linreg = LinearRegression()

pipe = make\_pipeline(column\_trans, linreg)

Run the cross validation 10 times using the negative mean absolute error as our scoring function. Finally, take the average of the 10 values to see the magnitude of the error:

cross\_val\_score(pipe, X, y, cv=10, scoring='neg\_mean\_absolute\_error').mean().round(2)

Which yields a value of -2937.17.

There is obviously much more analysis that can be done here but this is meant to illustrate how to use the scikit-learn functions in a more realistic analysis pipeline.

Conclusion

Encoding categorical variables is an important step in the data science process. Because there are multiple approaches to encoding variables, it is important to understand the various options and how to implement them on your own data sets. The python data science ecosystem has many helpful approaches to handling these problems. I encourage you to keep these ideas in mind the next time you find yourself analyzing categorical variables. For more details on the code in this article, feel free to review the [notebook](https://github.com/chris1610/pbpython/blob/master/notebooks/Category-Encoding-Article.ipynb).

…