

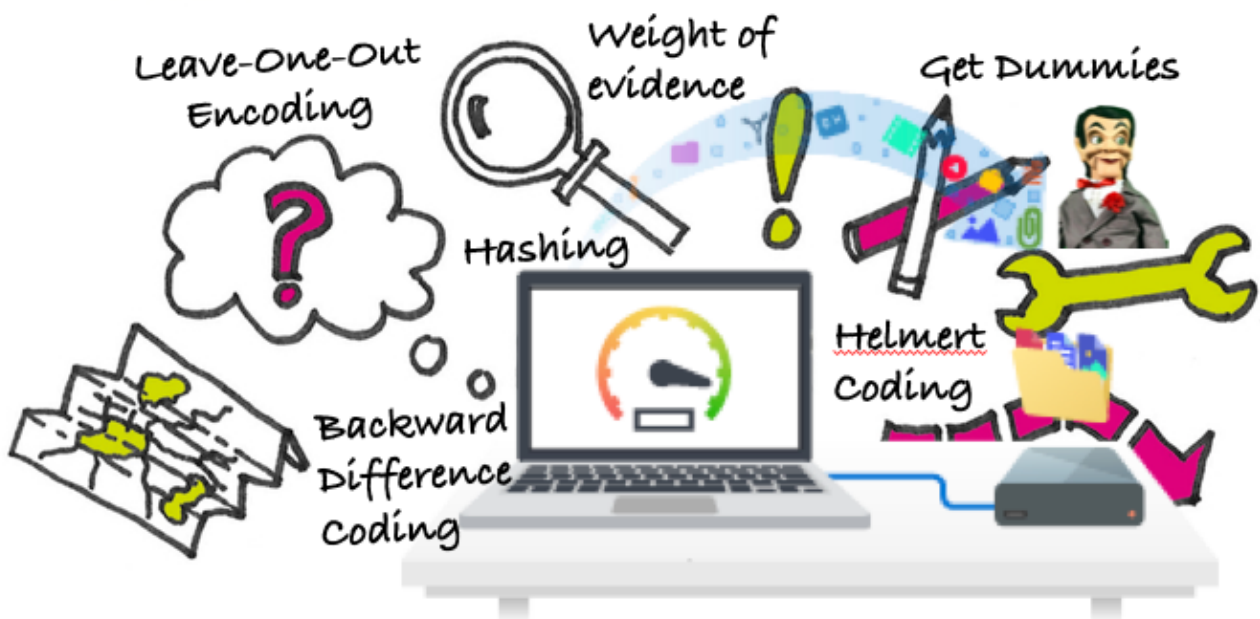
This is your **last** free story this month. Sign up and get an extra one for free.

A Data Scientist's Toolkit to Encode Categorical Variables to Numeric



Dr. Dataman

Dec 18, 2019 · 11 min read ★



Encoding categorical variables into numeric variables is part of a data scientist's daily work. I have been wanting to write down some tips for readers who need to encode categorical variables. The techniques in this article are the frequently used techniques in my professional work. I hope this article will assist you for any additional transformations to enhance your model performance. Below are the techniques:

- (A) One Hot Encoding
- (B) Weight of Evidence
- (C) Target Encoding

(D) Leave-One-Out Encoding

(E) Ordinal Encoding

(F) Hashing Encoding:

Because you are likely to produce data visualization exhibits for the new variables, it is worthwhile to take a look of my series of articles on data visualization, including “Pandas-Bokeh to Make Stunning Interactive Plots Easy”, “Use Seaborn to Do Beautiful Plots Easy”, “Powerful Plots with Plotly” and “Create Beautiful Geomaps with Plotly”. My goal in the data visualization articles is to assist you to produce data visualization exhibits and insights *easily* and *proficiently*. If you would like to adopt all these data visualization codes or make your work more proficient, take a look of them. I have written articles on a variety of data science topics. For the ease of use, you can bookmark my summary post “Dataman Learning Paths — Build Your Skills, Drive Your Career” that list the links to all articles.

(A) ONE-HOT ENCODING

Dummy encoding and one-hot encoding are the same thing; the former term comes from statistics and the latter from electrical engineering (electronics). Let me explain the subtle difference. Because a regression model can only take numeric variables, statistics has long solved the problem by converting a categorical variable of n values into $n-1$ dummy variables. Why $n-1$? This is to avoid the issue of multicollinearity (explained later). One-hot encoding converts a categorical variable of n values into n dummy variable. All the created variables have value 1 and 0. However, today's software lets you create all the dummy variables and let you decide which dummy variable to drop in order to prevent the multicollinearity issue.

There are many Python modules dealing with one-hot encoding. Here I present the `get_dummies` of Pandas and the `OneHotEncoder` of `category_encoders`. You can install the module `encode_category` via `pip install category_encoders`. To present you with a real case, I use the dataset Home Credit Default Risk from Kaggle to demonstrate the encoding methods. There are 18 categories plus the missing (NaN) for the variable “`occupation_type`” as shown below.

```
1 df = pd.read_csv('/application_train.csv')
2 df = df[['OCCUPATION_TYPE', 'OWN_CAR_AGE', 'TARGET']]
3 df['OCCUPATION_TYPE'].value_counts(dropna=False)
```

```

NaN          96391
Laborers     55186
Sales staff  32102
Core staff   27570
Managers     21371
Drivers      18603
High skill tech staff 11380
Accountants   9813
Medicine staff 8537
Security staff 6721
Cooking staff 5946
Cleaning staff 4653
Private service staff 2652
Low-skill Laborers 2093
Waiters/barmen staff 1348
Secretaries   1305
Realty agents 751
HR staff      563
IT staff      526
Name: OCCUPATION_TYPE, dtype: int64

```

(A.1) Get_dummy

```

1 #####
2 # Get_Dummy for One-Hot #
3 #####
4 dummies = pd.get_dummies(df['OCCUPATION_TYPE'], dummy_na=True)
5 dummies.head()
6 df2 = pd.concat([df, dummies], axis=1)
7 df2 = df2.drop(['Laborers'], axis=1)
8 df2.head()

```


get_dummies hosted with ❤ by GitHub

view raw

The dummy variables are created using

`pd.get_dummies(df['OCCUPATION_TYPE'], dummy_na=True)` . I then append the dummy variables back to the data. The results become:

	OCCUPATION_TYPE	TARGET
0	Laborers	1
1	Core staff	0
2	Laborers	0
3	Laborers	0
4	Core staff	0



Accountants Cleaning staff Cooking staff Core staff Drivers HR staff High skill tech staff IT staff Laborers Low-skill Laborers Managers Medicine staff Private service staff ...

0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
1	0	0	0	1	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	1	0	0	0	0
3	0	0	0	0	0	0	0	0	0	1	0	0	0	0
4	0	0	0	1	0	0	0	0	0	0	0	0	0	0

TIPS FOR YOU:

- Drop a level to avoid multicollinearity in a regression: Notice that I drop the dummy 'laborers'. Why do I drop it? If you are running a regression model, you need to drop the variable that shows multicollinearity with other variables. However, if you are running any tree-based algorithms, you do not need to drop the variable. You can find more explanations in my post "Avoid These Deadly Modeling Mistakes that May Cost You a Career".
- Choose the level of credible count to be the base level: The second question is why I drop 'laborers' but not other variables in my regression model? The criterion is *credibility*. In a regression the dropped value of the categorical variable becomes the base level that other values will reference to. If the count of the dropped value is too few, it is not good to be the base level that other levels reference to. For example, suppose you have a variable "state", and the first state in an alphabetical order is "AK" that has three observations. Many languages such as R use the first alphabetical level as the reference level, so the "AK" becomes the reference level. However, it has only three observations in your current dataset! It may be less than three observations if you repeat the same data drawing process. A better and common practice is to set the most frequent level, or a level that has sufficient number of observations.
- Do not forget the missing values: Notice that I use `dummy_na=True`. If you do not make the NAs explicitly as a dummy variable, you may forget it which will become the reference level in your regression. If the count of the NAs happens to be small, you will suffer the credibility issue as above. Also take a look of my post "Avoid These Deadly Modeling Mistakes that May Cost You a Career" for "what to do if a continuous variable has "NA", "0", "-99" or "-999".

(A.2) OneHotEncoder of Category_encoders

```

1 #####
2 # Category_Encoders for One-Hot #
3 #####
4 import category_encoders as ce
5
6 # MISSING VALUE TREATMENT

```

```
5 X = dt['OCCUPATION_TYPE']
6 ec = ce.OneHotEncoder(cols='OCCUPATION_TYPE', use_cat_names=True,
7     handle_unknown='indicator').fit(X)
8 ec.fit_transform(X)
```

ce_onehot hosted with ❤ by GitHub

[view raw](#)

I specify `handle_unknown='indicator'` to create the unknown category “OCCUPATION_TYPE_nan” which is the category for the missing values.

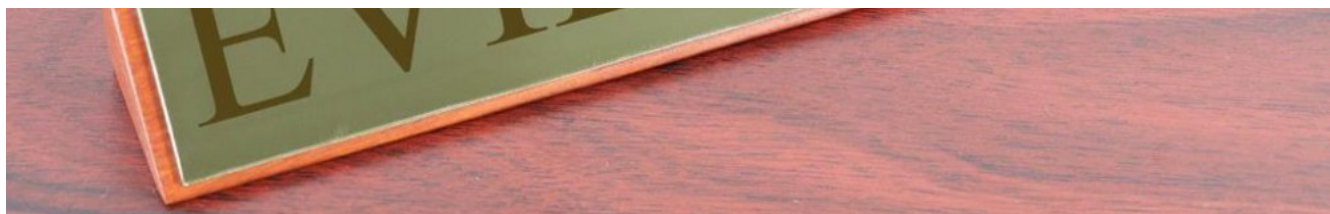
	OCCUPATION_TYPE_Laborers	OCCUPATION_TYPE_Core staff	OCCUPATION_TYPE_Accountants	OCCUPATION_TYPE_Managers	OCCUPATION_TYPE_nan	...
0	1	0	0	0	0	
1	0	1	0	0	0	
2	1	0	0	0	0	
3	1	0	0	0	0	
4	0	1	0	0	0	

The disadvantage of the one-hot or dummy encoding is it creates a very large sparse matrix if a categorical variable has many categories. For example, there are about 41,700 Zip Codes in the United States. If you convert the variable “Zip_code” using one-hot encoding, it will create a gigantic matrix, extremely inefficient, and prone to overfitting. Do we have better ways? Yes. Let’s look at the following techniques.

(B) WEIGHT OF EVIDENCE (WOE) ENCODING

Weight of evidence (WOE) is a widely used technique in *credit risk modeling* or called the *probability of default modeling* (such models predict how capable an applicant is able to repay the loan).





The goal of such transformation is to get the maximum difference among the binned categories relating to the target variable. It counts the number of responders and non-responders in each binned categories, then assigns a numeric value to each of the binned categories. In this transformation the information of the target variable has been utilized. When you have a categorical variable with many categories, WOE is a good choice. The WOE transformation can be extended to continuous dependent variable. Because of its valuable applications, I describe the two scenarios in (B.1) and (B.2).

(B.1) WOE with Binary Target Variable

In probability of default models the target values are either good or bad (default). So here I just adopt the popular WOE formula that still uses the term “good” or “bad” as the following:

$$WOE = \ln\left(\frac{\% \text{ Good}}{\% \text{ Bad}}\right) \cdot 100$$

Below I write a short Python function to calculate WOE. First, I create a “NoData” category to make sure missing values are counted as a category. For each category I count the total records and the number of “good” records. I then derive the percentage of the “good” and “bad” records. The WOE value is the odd ratio between the “good” and “bad” percentages.

```

1 #####
2 # My Function for WOE #
3 #####
4 def WOE(var):
5     df[var] = df[var].fillna('NoData')
6     k = df[[var, 'TARGET']].groupby(var)['TARGET'].agg(['count', 'sum']).reset_index()
7     k.columns = [var, 'Count', 'Good']
8     k['Bad'] = k['Count'] - k['Good']
9     k['Good %'] = (k['Good'] / k['Good'].sum()*100).round(2)
10    k['Bad %'] = (k['Bad'] / k['Bad'].sum()*100).round(2)
11    k[var+'_WOE'] = np.log(k['Good %'] / k['Bad %']).round(2)

```

```

12     k = k.sort_values(by=var+'_WOE')
13     return(k)
14     k = WOE('OCCUPATION_TYPE')
15     k

```

WOE_code hosted with ❤ by GitHub

[view raw](#)

Figure (1) shows the WOE for the variable “occupation_type”. Notice it has been ranked in ascending order.


	OCCUPATION_TYPE	Count	Good	Bad	Good %	Bad %	OCCUPATION_TYPE_WOE	
0	Accountants	9813	474	9339	1.91	3.30	-0.55	
6	High skill tech staff	11380	701	10679	2.82	3.78	-0.29	
10	Managers	21371	1328	20043	5.35	7.09	-0.28	
3	Core staff	27570	1738	25832	7.00	9.14	-0.27	
5	HR staff	563	36	527	0.15	0.19	-0.24	
12	NoData	96391	6278	90113	25.29	31.88	-0.23	
13	Private service staff	2652	175	2477	0.70	0.88	-0.23	
11	Medicine staff	8537	572	7965	2.30	2.82	-0.20	
7	IT staff	526	34	492	0.14	0.17	-0.19	
16	Secretaries	1305	92	1213	0.37	0.43	-0.15	
14	Realty agents	751	59	692	0.24	0.24	0.00	
1	Cleaning staff	4653	447	4206	1.80	1.49	0.19	
15	Sales staff	32102	3092	29010	12.46	10.26	0.19	
2	Cooking staff	5946	621	5325	2.50	1.88	0.29	
8	Laborers	55186	5838	49348	23.52	17.46	0.30	
17	Security staff	6721	722	5999	2.91	2.12	0.32	
18	Waiters/barmen staff	1348	152	1196	0.61	0.42	0.37	
4	Drivers	18603	2107	16496	8.49	5.84	0.37	
9	Low-skill Laborers	2093	359	1734	1.45	0.61	0.87	

Figure (1)

I append the WOE value of each category back to the original data. The first ten records are printed out in Figure (2).

```

1 df2 = pd.merge(df[['TARGET', 'OCCUPATION_TYPE']], k[['OCCUPATION_TYPE', 'WOE']],
2             left_on='OCCUPATION_TYPE',
3             right_on='OCCUPATION_TYPE', how='left')
4 df2.head(10)

```

WOE_head hosted with ❤ by GitHub

[view raw](#)

You also can use the function `WOEEncoder()` of the module `category_encoders` to calculate WOE as well. Below is how you perform the WOE transformation. You will find the WOE values are the same in Figure (2) either using my function or

`WOEEncoder()` .

```

1 #####
2 # Category_Encoders WOE #
3 #####
4 ec = ce.WOEEncoder()
5 df['OCCUPATION_TYPE'] = df['OCCUPATION_TYPE'].fillna('NoData')
6 X = df['OCCUPATION_TYPE']
7 y = df['TARGET']
8 X_WOE = ec.fit(X, y)
9 X_cleaned = ec.transform(X)
10 X_cleaned.round(2)

```

WOE_CE hosted with  by GitHub[view raw](#)

Using WOE()				ec = ce.WOEEncoder()	
TARGET	OCCUPATION_TYPE	OCCUPATION_TYPE_WOE		OCCUPATION_TYPE	
0	1	Laborers	0.30	0	0.30
1	0	Core staff	-0.27	1	-0.27
2	0	Laborers	0.30	2	0.30
3	0	Laborers	0.30	3	0.30
4	0	Core staff	-0.27	4	-0.27
5	0	Laborers	0.30	5	0.30
6	0	Accountants	-0.55	6	-0.55
7	0	Managers	-0.28	7	-0.28
8	0	NoData	-0.23	8	-0.23
9	0	Laborers	0.30	9	0.30

Figure (2)

What if the target is a continuous variable? You can split the predictor into 10 or 20 equal parts, then compute the WOE accordingly.

TIPS FOR YOU:

- Especially suitable for logistic regression: logistic regression fits a linear regression equation of predictors to predict the *logit-transformed* binary *Goods/Bads* target variable. The *Logit* transformation is simply the log of the odds. So by using the WOE-transformed predictors in logistic regression, the predictors are coded to the same WOE scale, and the parameters in the linear logistic regression equation can be directly compared.
- Remember to create the 'NoData' value for missing values
- Monotonic relationship with the target: between the target variable and the WOE-transformed variable. WoE transformation is strictly linear with respect to log odds

of the response with the unity correlation.

- There is no need to cap or floor the outliers: Given the binned categories, outliers will fall into the binned categories. The WOE value for each bin is the distribution of the good to that of bad, the concern for outliers disappear.

(B.2) WOE for Continuous Target Variable

Suppose now the target variable is the total income in the data, and we want to regress total income on the “occupation_type”. Because the target variable is a continuous variable, we will modify the WOE formula to be:

$$WOE = \ln\left(\frac{\% \text{ income}}{\% \text{ records}}\right) \cdot 100$$

Our continuous-target WOE Python function becomes:

```
1 #####
2 # My Function for Continous-Target WOE #
3 #####
4 def WOE_continuous(var,target):
5     df[var] = df[var].fillna('NoData')
6     k = df[[var,target]].groupby(var)[target].agg(['count','sum']).reset_index()
7     k.columns = [var,'Count','Sum']
8     k['Sum %'] = (k['Sum'] / k['Sum'].sum()*100).round(2)
9     k['Count %'] = (k['Count'] / k['Count'].sum()*100).round(2)
10    k[var+'_WOE'] = np.log(k['Sum %'] / k['Count %']).round(2)
11    k = k.sort_values(by=var+'_WOE')
12    return(k)
13 k = WOE_continuous('OCCUPATION_TYPE','AMT_INCOME_TOTAL')
14 k
```

WOE_sum hosted with ❤ by GitHub

[view raw](#)

	OCCUPATION_TYPE	Count	Sum	Sum %	Count %	OCCUPATION_TYPE_WOE	
1	Cleaning staff	4653	6.085700e+08	1.17	1.51	-0.26	■
9	Low-skill Laborers	2093	2.788462e+08	0.54	0.68	-0.23	■
2	Cooking staff	5946	8.229056e+08	1.59	1.93	-0.19	■
18	Waiters/barmen staff	1348	1.944794e+08	0.37	0.44	-0.17	■
17	Security staff	6721	1.005883e+09	1.94	2.19	-0.12	■
11	Medicine staff	8537	1.278071e+09	2.46	2.78	-0.12	■
15	Sales staff	32102	4.889227e+09	9.42	10.44	-0.10	■
12	NoData	96391	1.479756e+10	28.51	31.35	-0.09	■
16	Secretaries	1305	2.095069e+08	0.40	0.42	-0.05	■
8	Laborers	55186	9.180604e+09	17.69	17.95	-0.01	
3	Core staff	27570	4.760145e+09	9.17	8.97	0.02	■

6	High skill tech staff	11380	2.080742e+09	4.01	3.70	0.08	■
13	Private service staff	2652	4.835519e+08	0.93	0.86	0.08	■
4	Drivers	18603	3.478977e+09	6.70	6.05	0.10	■
5	HR staff	563	1.063599e+08	0.20	0.18	0.11	■
0	Accountants	9813	1.909397e+09	3.68	3.19	0.14	■
14	Realty agents	751	1.464480e+08	0.28	0.24	0.15	■
7	IT staff	526	1.122829e+08	0.22	0.17	0.26	■
10	Managers	21371	5.563655e+09	10.72	6.95	0.43	■

Figure (3)

Figure (3) shows the WOE for continuous variable. When I append the WOE values back to the original data, the first five records have the following values:

	TARGET	OCCUPATION_TYPE	OCCUPATION_TYPE_WOE
0	1	Laborers	-0.01
1	0	Core staff	0.02
2	0	Laborers	-0.01
3	0	Laborers	-0.01
4	0	Core staff	0.02

(C) TARGET ENCODING

The targeting encoding simply replaces each category with the mean target value for samples which have that category. The code below assigns the mean value of a category to the observations of a category.

```

1 #####
2 # Category_encoders for target encoding #
3 #####
4 from category_encoders import target_encoder as te
5 df['OCCUPATION_TYPE'] = df['OCCUPATION_TYPE'].fillna('NoData')
6 X = df['OCCUPATION_TYPE']
7 y = df['TARGET']
8 ec = te.TargetEncoder()
9 X_TE = ec.fit_transform(X,y)
10 outf = pd.concat([X,X_TE],axis=1)
11 outf.columns = ['OCCUPATION_TYPE','mean']
12 outf.head(10)

```

TE hosted with  by GitHub

[view raw](#)

	OCCUPATION_TYPE	mean
0	Laborers	0.105788

1	Core staff	0.063040
2	Laborers	0.105788
3	Laborers	0.105788
4	Core staff	0.063040
5	Laborers	0.105788
6	Accountants	0.048303
7	Managers	0.062140
8	NoData	0.065131
9	Laborers	0.105788

TIPS FOR YOU:

- Targeting encoding can be seriously plagued by overfitting because it uses information about the target, called *target leakage*.
- To overcome this issue, you can add tiny random values to the new variable in order not to be fit too precisely for the training data. You may ask how tiny is tiny? The code below generates noises from 0 to the capped value which is a fraction (=0.3) of the mean of the target (you can compare the first ten records with those above). This fraction `cntrl` controls the size of the random values. How do you determine this fraction? In your modeling process you experiment with different values. You shall settle with a value that resulting the best performance with your test data.
- When you predict new records with your model, you do not need to add random values to the new variable.

```

1 #####
2 # Category_encoders for target encoding #
3 #####
4 from category_encoders import target_encoder as te
5 df['OCCUPATION_TYPE'] = df['OCCUPATION_TYPE'].fillna('NoData')
6 X = df['OCCUPATION_TYPE']
7 y = df['TARGET']
8 ec = te.TargetEncoder()
9 X_TE = ec.fit_transform(X,y)
10 outf = pd.concat([X,X_TE],axis=1)
11 outf.columns = ['OCCUPATION_TYPE','mean']
12
13 # add some noises
14 cntrl = 0.3
15 capped = df['TARGET'].max() * cntrl

```

```

15 capped = ut[ 'TARGET' ].mean() * cutoff
16 num_obs = df.shape[0]
17 noise = np.random.uniform(0,capped,num_obs)
18 outf['mean'] = outf['mean'] + noise
19 outf.head(10)

```

TE_2 hosted with ❤ by GitHub

[view raw](#)

	OCCUPATION_TYPE	mean
0	Laborers	0.123572
1	Core staff	0.076551
2	Laborers	0.108224
3	Laborers	0.122943
4	Core staff	0.077079
5	Laborers	0.124371
6	Accountants	0.055527
7	Managers	0.081422
8	NoData	0.072583
9	Laborers	0.124686

(D) LEAVE-ONE-OUT (LOO) ENCODING

Do not confuse Leave-One-Out (LOO) Encoding with Leave-One-Out Cross Validation (LOOCV). LOOCV is a model validation technique in assessing how your machine learning model will generalize to an independent data set.

As discussed in (C), target leakage — using information about the target, can result in seriously overfitting issue. To overcome such challenge, Leave-One-Out Encoding produces the mean value of the target over all rows for the same categorical level, excluding the row itself. It leaves the target value of the row itself out — thus the name *Leave-One-Out*. This avoids direct target leakage.

I will first use the leave-one-out function in category_encoders to show you how easily it can be done. Then I write down my own function to walk you through how it is generated.

	OCCUPATION_TYPE_LOO
0	0.11
1	0.06
2	0.11
3	0.11

0	1	2	3	4
0.11				0.06

Figure (4)

The “my function” below first get the count and the sum statistics by category, then append the statistics back to the data. Now the key step is Line 12 where it subtracts the target value of the row itself to get the mean.

```

1 #####
2 # My Function for Leave-One-Out #
3 #####
4 def LOO(var,target):
5     # Get the count and the sum statistics by category
6     df['OCCUPATION_TYPE'] = df['OCCUPATION_TYPE'].fillna('NoData')
7     h = df[['OCCUPATION_TYPE','TARGET']].groupby('OCCUPATION_TYPE')['TARGET'].agg(['count','sum'])
8     h.columns = ['OCCUPATION_TYPE','Count','Sum']
9     # Append to the data
10    df2 = pd.merge(df[[var,target]],h,left_on='OCCUPATION_TYPE',right_on='OCCUPATION_TYPE',how='left')
11    # Get the mean excluding the row itself to avoid direct target leakage
12    df2[var + '_LOO'] = ((df2['Sum'] - df2[target])/(df2['Count'] - 1)).round(2)
13    df2 = df2.drop([target,'Count','Sum'],axis=1)
14    return(df2)
15
16 k = LOO('OCCUPATION_TYPE','TARGET')
17 k.head()

```

LOO hosted with ❤ by GitHub

[view raw](#)

	OCCUPATION_TYPE	OCCUPATION_TYPE_LOO
0	Laborers	0.11
1	Core staff	0.06
2	Laborers	0.11
3	Laborers	0.11
4	Core staff	0.06

Figure (5)

The leave-one-out score in Figure (5) are the same as that in Figure (4).

(E) ORDINAL ENCODING

Ordinal encoding uses additional information to convert the category variable to numeric. A good example is education level as shown in Figure (6). It can be coded by a label (EDU_1), or by the number of years in completing the level. The `category_encoders` module enables you to do so.

Level	EDU_1	EDU_2
Elementary	1	6
High School	2	12
2-year college	3	14
College graduate	4	18
Graduate degree	5	20

Figure (6)

TIPS FOR YOU:

- The numeric differences matter. So you need to consider if the numbers make sense and your interpretation is consistent.

(F) HASHING ENCODING

If you use one-hot encoding to convert a high-cardinality categorical variable (such as zip codes), you will end up with a gigantic sparse matrix. Such sparse matrix is extremely inefficient and will result in model instability. The hashing encoding may be a better solution. Hashing encoding gains its popularity in the discussions of Kaggle competitions. It is similar to one-hot encoding but projects to much less number of dimensions. Hashing encoding uses a hash function to convert N levels to $M \leq N$ columns.



How does it work? Let me use Figure (H) to walk you through the details. First, each of your categorical values is converted to a hexadecimal value through the MD5 hash function. The MD5 algorithm is a widely used hash function. You can play with it by using this MD5 Hash Generator. Second, the hexadecimal values are converted to the decimal values, which you can mimic through this Hexadecimal to Decimal Converter. Then the decimal values are divided by the column length (the code example below uses 4) to get the remainders. You can use this Long Division Calculator to get the remainders. Finally the remainders indicate which column to turn on or off (1 or 0). Notice that there is no way you can do a reverse lookup to determine what the input was. So hashing is not *reversible*.

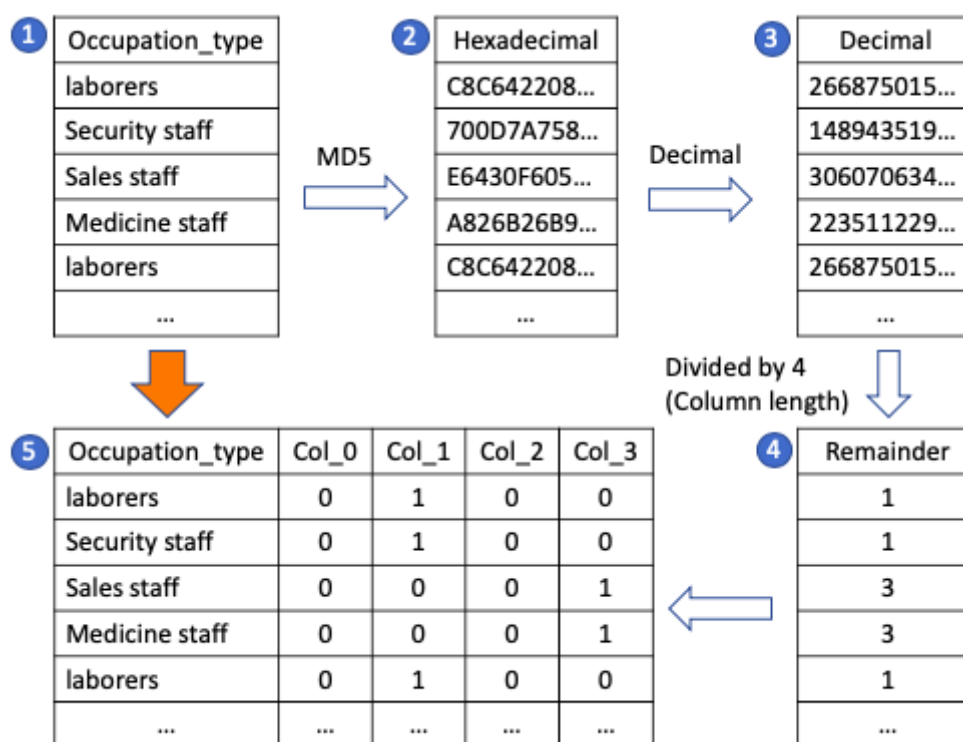


Figure (7): Hashing Encoding Process

Hashing encoding can be done easily by the module `category_encoders`. Below shows the code. I set `n_components=4` so the results have four columns. The default value is `n_components=8`. If you set `n_components` to be the same as the number of levels, the hashing encoding becomes the one-hot encoding.

```
1 #####
2 # Category_encoders for hashing encoding #
3 #####
4 from category_encoders import hashing as hs
```

```

5 df['OCCUPATION_TYPE'] = df['OCCUPATION_TYPE'].fillna('NoData')
6 X = df['OCCUPATION_TYPE']
7 ec = hs.HashingEncoder(n_components=4,hash_method='md5')
8 X_HASH = ec.fit_transform(X)
9 outf = pd.concat([X,X_HASH],axis=1)
10 outf.head()

```

hashing hosted with ❤ by GitHub

[view raw](#)

	OCCUPATION_TYPE	col_0	col_1	col_2	col_3
0	Laborers	0	1	0	0
1	Core staff	0	0	0	1
2	Laborers	0	1	0	0
3	Laborers	0	1	0	0
4	Core staff	0	0	0	1

What are the categorical levels assigned to each column? I do a `value_counts()` for each column and show the results below. Col_0 only has one value “NoData”, Col_1 has seven levels, and so on. This coding logic is determined by the process in Figure (7). This encoding process collapses some levels to the same outcome, called *collisions*. Although some information is lost, interestingly, the collisions do not significantly affect performance unless there is a great deal of overlap.

```

1 outf.loc[outf['col_0']==1,'OCCUPATION_TYPE'].value_counts()
2 outf.loc[outf['col_1']==1,'OCCUPATION_TYPE'].value_counts()
3 outf.loc[outf['col_2']==1,'OCCUPATION_TYPE'].value_counts()
4 outf.loc[outf['col_3']==1,'OCCUPATION_TYPE'].value_counts()

```

value_counts hosted with ❤ by GitHub

[view raw](#)

Col_0	Count
NoData	96391

Col_1	Count
Managers	21371
Drivers	18603
Accountants	9813
Security staff	6721
Private service staff	2652
Realty agents	751
HR staff	563

Col_2	Count
Cleaning staff	4653
Low-skill Laborers	2093
Waiters/barmen staff	1348

Col_3	Count
Core staff	27570
Medicine staff	8537
Cooking staff	5946
Secretaries	1305
IT staff	526

TIPS FOR YOU:

- The disadvantage of the Hashing encoding is the lack of good business interpretation. Why do “Managers”, “Drivers”, ..., “HR staff” all go to the same group?
- The process can be time-consuming.
- If you want to have a more interpretable result, you can group categorical levels manually.

• • •

A few other encoding techniques such as CatBoost Encoding, Polynomial Encoding, Sum Encoding, or James-Stein Encoding are not covered here. It is because the above techniques are sufficient to most of my professional work, or the uncovered techniques are some variations of the techniques. Readers are encouraged to check the `category_encoders` for more reference.

[Data Science](#) [Machine Learning](#) [Artificial Intelligence](#)

[About](#) [Help](#) [Legal](#)

Get the Medium app

