CAPTER – 5

Encoding Techniques

*Learning Topics*



* What is encoding
* One-hot / dummy encoding
* Ordinal encoding
* Nominal and ordinal data
* Types of encoding technique
* Duplicate values.
* Access subset of data
* Boolean indexing

**GitHub link:** [*Encoding\_Techniques*](https://github.com/ramasureshvijjana/Data_Science/tree/master/05_Dealing_With_Categorical_Data)

Getting started in applied machine learning can be difficult, especially when working with real-world data. Often, machine learning tutorials will recommend or require that you prepare your data in specific preprocessing ways before fitting a machine learning model. In this chapter will learn one of the preprocessing step called *Label Encoding*.

**1. Encoding:**

* Encoding is the process of converting string / labeled data (categorical data) into numerical data.

**1.1. What is Categorical Data:**

* Categorical data are variables that contain label / string values rather than numeric values.
* *Some examples include:*
* A “*pet*” variable with the values: “*dog*” and “*cat*“.
* A “*color*” variable with the values: “*red*“, “*green*” and “*blue*“.
* A “*place*” variable with the values: *“first”,* “*second*” and “*third*“.
* Here each value represents a different category.

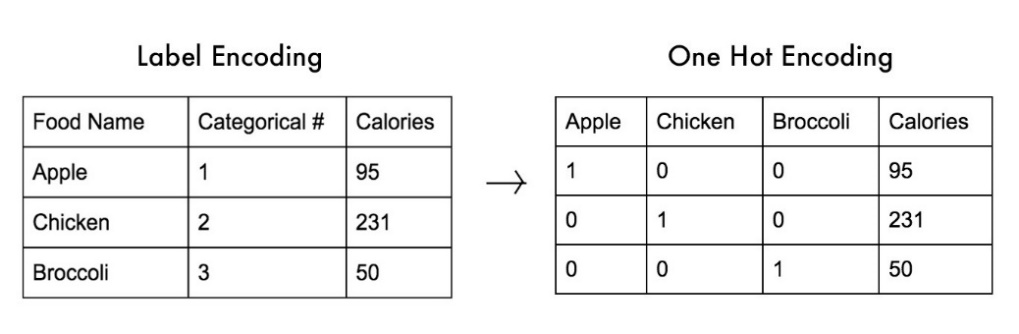
## 1.2. What is the Problem with Categorical Data:

* Some algorithms can work with categorical data directly. For example, a decision tree can be learned directly from categorical data, no data transform required.
* Many machine learning algorithms cannot operate on label data directly. They require all input variables and output variables to be numeric.
* This means that categorical data must be converted to a numerical form.

**1.3. Types of Encoding Technique:**

There are plenty of methods to encode categorical variables into numeric and each method comes with its own advantages and disadvantages. To discover them, we will see the following ways to encode categorical variables:

1. One-hot/dummy encoding
2. Binary encoding
3. Label encoding
4. Ordinal encoding
5. Frequency / count encoding
6. Target encoding / Mean Encoding
7. Feature Hashing
8. Weight of evidence encoding

**2. One-Hot / Dummy Encoding:**

In this technique, the categorical parameters will prepare as a separate column for each label and assign a value 1 or 0 based on its presence in that particular row.

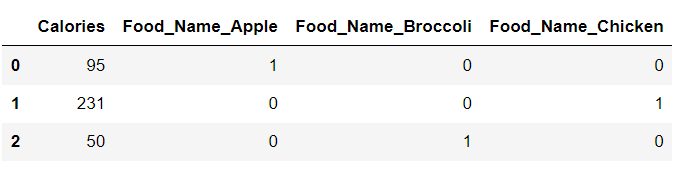
Let’s understand it with an example, consider the data where foods (apple, chicken, broccoli) and their corresponding calories are given. These food names need to convert into numerical by using one hot encoding.

* First it will take all unique labels from “*Food Name”* column and creates each column for each label.
* Now three labels Apple, chicken, and broccoli are three columns in the new data set and each column assigned with a value 1or 0 as shown in above image.

**2.1. Code:**

|  |
| --- |
| 1. import pandas as pd 2. import numpy as np 3. from sklearn.preprocessing import OneHotEncoder 4. # One hot encoding 5. data\_1 = pd.get\_dummies(data, columns=['Food\_Name']) 6. display(data\_1) 7. ####################### WAY -2 ############################# 8. ## Code with sklearn package by using OneHotEncoder() 9. #Create an instance of One-hot-encoder 10. enc=OneHotEncoder() 11. '''NOTE: we have converted the enc.fit\_transform() method to array because the fit\_transform method of OneHotEncoder returns SpiPy sparse matrix this enables us to save space when we have huge  number of categorical variables''' 12. columns = sorted(data['Food\_Name'].unique()) 13. print(f'Food\_Name labels: {columns}') 14. enc\_arr = enc.fit\_transform(data[['Food\_Name']]).toarray() 15. enc\_data = pd.DataFrame(enc\_arr, columns=columns).astype(int) 16. New\_df=data.join(enc\_data).drop(['Food\_Name'], axis=1) 17. print(New\_df) |

**Output:**



**2.2. Advantages:**

* ***Simplicity:*** One-hot encoding is a straightforward method. It easier to understand the relationship between the categorical variables and the target variable.
* ***Compatibility:*** One-hot encoding is compatible with most machine learning algorithms.
* ***Avoiding bias:*** One-hot encoding helps to avoid the bias that may be introduced by encoding categorical values as ordinal values.
* ***Better performance***: One-hot encoding often results in improved performance of machine learning models, particularly with decision trees and Random Forest algorithms.
* ***Efficient storage:*** One-hot encoding is space-efficient, as the encoded data can be stored as a sparse matrix, which uses less memory than dense matrices.

#### **2.3. Disadvantages:**

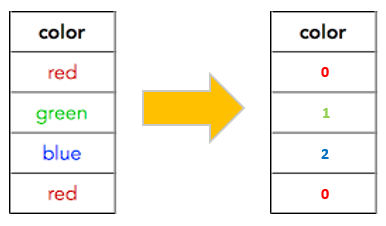
* It can lead to increased dimensionality, as a separate column for each label. This can make the model more complex and slower to train.
* It can lead to sparse data, as most observations will have a value of 0 in most of the one-hot encoded columns. These 0 values create negligible / infinity values in mathematical calculations in some models (ANN, Gradient Decent, etc...)
* It can lead to overfitting, especially if there are many categories in the variable and the sample size is relatively small.
* It can lead the multi-correlation problem due to high dependency between one-hot encoding columns.

**3. Label encoding:**

Label encoding is a process of converting categorical data into numerical data by assigning a unique integer label to each category. but it does not add any additional information to the data.

The steps involved in label encoding are as follows:

1. Identify the categorical variable that needs to be encoded.
2. Assign a unique numerical label to each category of the variable, starting from 0 or 1.
3. Replace the categorical values in the dataset with their corresponding numerical labels.
4. Store the mapping of the original categorical values to their numerical labels, as it may be needed later for decoding the labels back to their original values.

For example, let's say we have a categorical variable "color" with three categories: "red", "green", and "blue". To encode this variable using label encoding, we would assign the labels 0, 1, and 2 to the categories respectively. The resulting encoded data would replace "red" with 0, "green" with 1, and "blue" with 2. We would also store the mapping of the original categories to their numerical labels, as follows:

{"red": 0, "green": 1, "blue": 2}

This mapping may be useful later for decoding the labels back to their original values.

**3.1. Code:**

|  |
| --- |
| 1. # Import label encoder 2. from sklearn import preprocessing 3. # label\_encoder object knows 4. # how to understand word labels. 5. label\_encoder = preprocessing.LabelEncoder() 6. # Encode labels in column 'species'. 7. df['color']= label\_encoder.fit\_transform(df['color']) 8. print(df) |

**Output:**

|  |  |
| --- | --- |
| ***Before label encoder:*** | ***After label encoder:*** |

**3.2. Advantages:**

* ***Simplicity:*** Label encoding is a simple technique that is easy to implement and understand.
* ***Space efficiency:*** Label encoding typically requires less memory than one-hot encoding, another common technique for encoding categorical data.
* ***Maintains the range of the variable:*** Unlike one-hot encoding, which can expand the feature space substantially, label encoding keeps the range of the variable within a reasonable size.

#### **3.3. Disadvantages:**

* ***Arbitrary numerical assignments:*** Label encoding assigns *arbitrary numerical values* to categorical variables, which may not represent the true relationship between the categories. This can result in misleading results and reduced accuracy.
* ***Implies order:*** Label encoding implies an order between categories, even when there is none. For example, encoding "blue" as 2 and "green" as 1 implies that blue is somehow "better" or "higher" than green, which is not true.
* ***May not be suitable for nominal variables:*** Label encoding may not be suitable for nominal variables, which do not have any inherent order or hierarchy between the categories.
* ***Sensitive to initial assignment:*** The assigned numerical values are sensitive to the initial assignment and can lead to different results for the same data set if assigned differently.
* ***May lead to overfitting:*** Label encoding may lead to overfitting if the numerical values assigned to the categories are too similar or too different, as the algorithm may learn the encoding itself rather than the actual data.

**4. Ordinal Encoding:**

* Ordinal encoding technique using when categorical data is ordinal data.

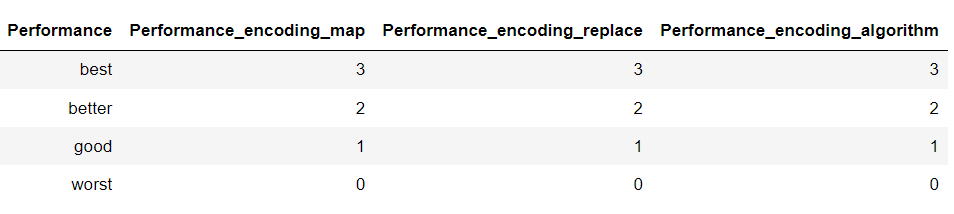
|  |  |
| --- | --- |
| ***Label*** | ***Encoding value*** |
| Worst | 0 |
| Good | 1 |
| Better | 2 |
| Best | 3 |

* Whenever your categorical feature has ranks in between then the labels then that feature is called the ordinal data. The encoding values should be assigned based on the priority in this kind of data.
* For example, data have a feature called *performance* and it contains labels as *good, best, better, worst*. These all labels have some priority between them. Observe that priority in below.

**4.1. Code:**

|  |
| --- |
| 1. from sklearn.preprocessing import OrdinalEncoder 2. import pandas as pd 3. import numpy as np 4. # Creating distinct label values for mapping purpose. 5. encoders = {"best": 3, "better": 2, "good":1, "worst":0} 6. labels\_order = ["worst", "good", "better", "best"] 7. # Way-1: Label Encoding Using map() Function 8. data["Performance\_encoding\_map"] = data['Performance'].map(encoders) 9. # Way-2: Label Encoding Using replace() Function 10. data["Performance\_encoding\_replace"] = data['Performance'].replace(encoders) 11. # Way- 3: Label Encoding Using sklearn Librery 12. oe = OrdinalEncoder(categories=[labels\_order], dtype=int) 13. data["Performance\_encoding\_algorithm"] = oe.fit\_transform(np.array(data["Performance"]).reshape(-1,1)) 14. print(oe.categories\_) 15. data |

**Output:**

****

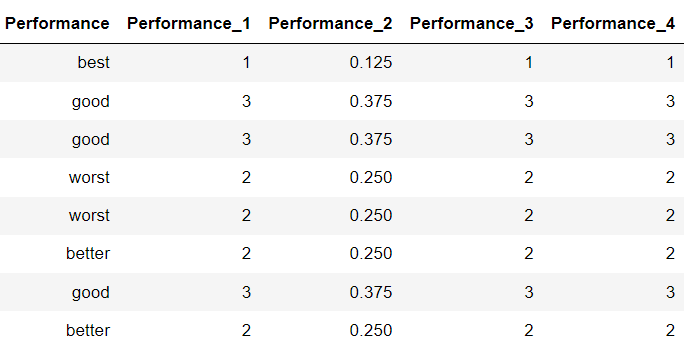
**5. Frequency or Count Encoding:**

* Frequency encoding Converts label to numerical based on the count of a label in a column.
* For example, if we have a column “*city\_name*” and it contains three Labels are *India,* *America,* *South Korea*. Let’s consider *India* appears 10 times, *America* appears 9 times, and *South Korea* appears 11 times. In this case the *India* will replace with 10 and *America* replaced with 9 and *South Korea* replaced with 11.

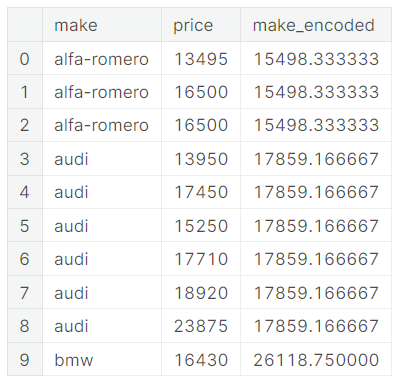
**5.1. Code:**

|  |
| --- |
| 1. from feature\_engine.encoding import CountFrequencyEncoder 2. import category\_encoders  as ce 3. import pandas as pd 4. # Way - 1: Using map() 5. performance\_frq\_dict = data['Performance'].value\_counts().to\_dict() 6. data['Performance\_1'] = data['Performance'].map(performance\_frq\_dict) 7. data 8. # Way - 2 : Using groupby() 9. performance\_frq\_dict = dict(data.groupby('Performance').size()/len(data)) 10. data['Performance\_2'] = data['Performance'].map(performance\_frq\_dict) 11. data 12. # Way - 3 : Using CountEncoder 13. count\_encoder = ce.CountEncoder(cols=['Performance']) 14. data['Performance\_3'] = count\_encoder.fit\_transform(data['Performance']) 15. data 16. # Way - 4 : Using CountFrequencyEncoder 17. count\_frequency\_encoder = CountFrequencyEncoder() 18. data['Performance\_4'] = count\_frequency\_encoder.fit\_transform(data[['Performance']]) 19. data |

**Output:**



**6. Target or Mean Encoding:**

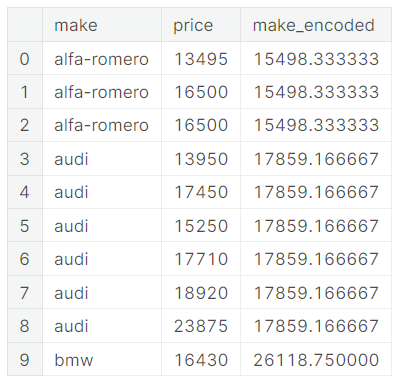
***Def:*** A target encoding is any kind of encoding that replaces a feature's categories with some number derived from the target.

* The target encoding required two features. one is encodable feature called input feature, and second one is target feature.
* Here the target feature should affect the input feature. which means input feature should have dependency on target feature.
* Let’s take a small data to understand the it better:

Here *Make* = Input feature

*Price* = Target feature

**6.1. Target Encoding Using Mean Technique:**



In feature mean calculation for aifa-romero label**:**

We can see all of each label in right table. These mean values are encoded values.

There is a problem involved in above mean technique, those are:

1. If anyone of the label appears very less time in that feature, then that label should have low mean value. But it generates accurate mean that can be high value if we compared with high frequency labels.

A solution to these problems is applying smoothing technique as explained below.

**6.2. Target Encoding Using Smoothing Technique:**

* The idea of smoothing technique is to blend the with the .
* Rare categories get less weight on their category average, and the missing categories just get the overall average.

Formula:

= averages value of all target values

*w* = weight (it is between 0 and 1)

*n* = total number of times that category occurs in the data

*m* = "smoothing factor". Larger values of m put more weight on the overall estimate.

Importance of *m* value:

* The selection of *m* value is very important in above calculation.
* When the distance or difference between (noise of the target data) the target values is more of a particular label. Then, we required mode data to concluded the stable mean. In this condition, m values should be more.
* When the distance or difference (noise of the target data) between the target values is low of a particular label. Then, we concluded the stable mean very easily. In this condition, less m value is sufficient.

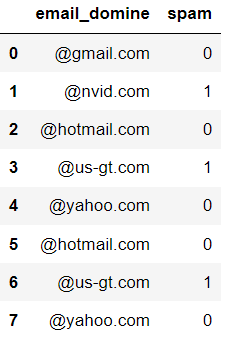
**6.3. Code:**

|  |
| --- |
| 1. from sklearn.preprocessing import OrdinalEncoder 2. import pandas as pd 3. import numpy as np 4. # Creating distinct label values for mapping purpose. 5. encoders = {"best": 3, "better": 2, "good":1, "worst":0} 6. labels\_order = ["worst", "good", "better", "best"] |

**Output:**

**7. Feature Hashing Encoding:**

***Def:*** Feature hashing is a technique used in machine learning to transform categorical data into a numerical format that can be used in models.



**7.1. Woking:**

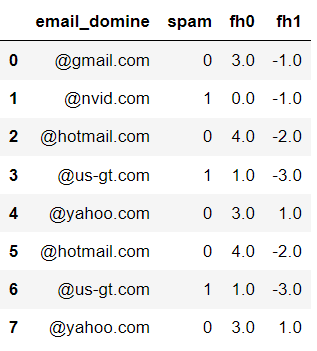
* Let’s consider we have a dataset of emails as shown in below, and we want to predict the email is spam or not?
* Now we need to convert the “email\_domine” column into numerical by using feature hashing encoding technique.
* In feature hashing, first we should select the bins value and based on bins value feature hashing will assign a unique number to each text. The result dataset looks like below.

Note: If the bins value is less then number of unique labels in categorical data leads collisions issue i.e., different categories hashing to the same value.

**7.2. Code:**

|  |
| --- |
| 1. from sklearn.feature\_extraction import FeatureHasher 2. n\_features = 2 3. features = [f"fh{n}" for n in range(0, n\_features)] 4. fh = FeatureHasher(n\_features=n\_features, input\_type='string') 5. features\_array = fh.fit\_transform(data['email\_domine']) 6. hasher\_df = pd.DataFrame(r.toarray(), columns= features) 7. data = pd.concat([data, hasher\_df], axis=1) |

**Output:**

****

**8. Weight of Evidence (WOE) Encoding:**

***Def:*** The WOE value quantifies the relationship between a category and the target variable. It measures how well the category predicts the positive (1) or negative (0) class of the target variable.

In this technique each label will be replaced with WOE value calculated by using below formula:

Here = Number of nth labels belongs to +ve (1) class.

= Number of nth labels belongs to -ve (0) class.

= Number of +ve (1) classes in whole dataset.

= Number of -ve (0) classes in whole dataset.

**The WOE value can be positive or negative:**

* If WOE>0, it indicates that the category is associated with a higher likelihood of the positive event.
* If WOE<0, it indicates that the category is associated with a higher likelihood of the negative event.
* If WOE=0, it suggests that the category has no discriminatory power between the positive and negative events.

**When to Use WOE:**

1. Binary Classification Problems: WOE is most commonly used in binary classification problems where you have a binary target variable (0 or 1) and you want to assess the predictive power of categorical independent variables (features) on this binary target.
2. Categorical Variables: WOE is beneficial when dealing with categorical variables with multiple levels or categories. It helps transform these variables into a numeric form that can be directly used in machine learning models like logistic regression.
3. Feature Selection: WOE can be used as a feature engineering technique to select the most informative categories within a categorical variable. This helps reduce dimensionality and improve model performance.
4. Handling Missing Values: WOE can be used to handle missing values within categorical variables. You can create a separate category or bin for missing values and calculate its WOE.
5. Addressing Class Imbalance: When dealing with imbalanced datasets, especially in credit scoring or fraud detection, WOE can help capture the characteristics of the minority class effectively.
6. Collinearity: WOE can be a useful technique to address collinearity issues within categorical variables by grouping similar categories together based on their impact on the target.