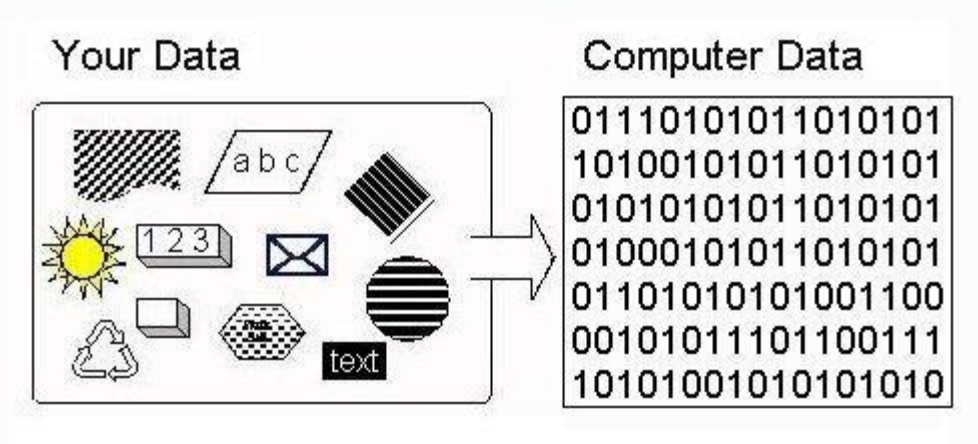
Table of Contents

What is Categorical Encoding?

Typically, any structured dataset includes multiple columns – a combination of numerical as well as categorical variables. A machine can only understand the numbers. It cannot understand the text. That's essentially the case with Machine Learning algorithms too.



That's primarily the reason we need to convert categorical columns to numerical columns so that a machine learning algorithm understands it. This process is called **categorical encoding**.

For exp gender Male,Female converts it to number 0,1

Pen,pencil,shoe converts it to number 0,1

> Categorical encoding is a process of converting categories to numbers.

In the next section, I will touch upon different ways of handling categorical variables.

Different Approaches to Categorical Encoding

So, how should we handle categorical variables? As it turns out, there are multiple ways of handling Categorical variables. In this article, I will discuss the two most widely used techniques:

- Label Encoding
- One-Hot Encoding

Now, let us see them in detail.

Label Encoding

Label Encoding is a popular encoding technique for handling categorical variables. In this technique, each label is assigned a **unique integer based on alphabetical ordering**.

Let's see how to implement label encoding in Python using the scikit-learn library and also understand the challenges with label encoding.

Let's first import the required libraries and dataset:

```
#importing the libraries
import pandas as pd
```

```
import numpy as np


#reading the dataset
df=pd.read_csv("Salary.csv")
```

view rawOne8.py hosted with by GitHub

Output:

Country	Age	Salary
India	44	72000
US	34	65000
Japan	46	98000
US	35	45000
Japan	23	34000

Understanding the datatypes of features:

```
print df.info
```

view rawone9.py hosted with by GitHub

Output:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 14 entries, 0 to 13
Data columns (total 3 columns):
Country    14 non-null object
Age        14 non-null int64
Salary     14 non-null int64
dtypes: int64(2), object(1)
memory usage: 416.0+ bytes
```

As you can see here, the first column, Country, is the categorical feature as it is represented by the **object data type** and the rest of them are numerical features as they are represented by *int64*.

Now, let us implement label encoding in Python:

```
# Import label encoder
from sklearn import preprocessing
# label_encoder object knows how to understand word labels.
```

```
label_encoder = preprocessing.LabelEncoder()
# Encode labels in column 'Country'.
data['Country']= label_encoder.fit_transform(data[‘Country'])
print(data.head())
```

view rawOne3.py hosted with by GitHub

Output:

Country	Age	Salary
0	44	72000
2	34	65000
1	46	98000
2	35	45000
1	23	34000

As you can see here, label encoding uses alphabetical ordering. Hence, India has been encoded with 0, the US with 2, and Japan with 1.

Challenges with Label Encoding

In the above scenario, the Country names do not have an order or rank. But, when label encoding is performed, the country names are ranked based on the alphabets. Due to this, there is a very high probability that the model captures the relationship between countries such as India < Japan < the US.

This is something that we do not want! So how can we overcome this obstacle? Here comes the concept of **One-Hot Encoding**.

One-Hot Encoding

One-Hot Encoding is another popular technique for treating categorical variables. **It simply creates additional features based on the number of unique values in the categorical feature**. Every unique value in the category will be added as a feature.

One-Hot Encoding is the process of creating dummy variables.

In this encoding technique, each category is represented as a one-hot vector. Let’s see how to implement one-hot encoding in Python:

```
# importing one hot encoder
from sklearn from sklearn.preprocessing import OneHotEncoder
# creating one hot encoder object
```

```
onehotencoder = OneHotEncoder()
#reshape the 1-D country array to 2-D as fit_transform expects 2-D and finally fit the
object
X = onehotencoder.fit_transform(data.Country.values.reshape(-1,1)).toarray()
#To add this back into the original dataframe
dfOneHot = pd.DataFrame(X, columns = ["Country_"+str(int(i)) for i in
range(data.shape[1])])
df = pd.concat([data, dfOneHot], axis=1)
#droping the country column
df= df.drop(['Country'], axis=1)
#printing to verify
print(df.head())
```

view rawOne4.py hosted with by GitHub

Output:

0	1	2	Age	Salary
1	0	0	44	72000
0	0	1	34	65000
0	1	0	46	98000
0	0	1	35	45000
0	1	0	23	34000

As you can see here, 3 new features are added as the country contains 3 unique values – India, Japan, and the US. In this technique, we solved the problem of ranking as each category is represented by a binary vector.

Can you see any drawbacks with this approach? Think about it before reading on.

Challenges of One-Hot Encoding: Dummy Variable Trap

One-Hot Encoding results in a Dummy Variable Trap as the outcome of one variable can easily be predicted with the help of the remaining variables.

Dummy Variable Trap is a scenario in which variables are highly correlated to each other.

The Dummy Variable Trap leads to the problem known as **multicollinearity**. Multicollinearity occurs where there is a dependency between the independent features. Multicollinearity is a serious issue in machine learning models like Linear Regression and Logistic Regression.

So, in order to overcome the problem of multicollinearity, one of the dummy variables has to be dropped. Here, I will practically demonstrate how the problem of multicollinearity is introduced after carrying out the one-hot encoding.

One of the common ways to check for multicollinearity is the Variance Inflation Factor (VIF):

- VIF=1, Very Less Multicollinearity
- VIF<5, Moderate Multicollinearity
- VIF>5, Extreme Multicollinearity (This is what we have to avoid)

Compute the VIF scores**:**

```python
# Function to calculate VIF
def calculate_vif(data):
    vif_df = pd.DataFrame(columns = ['Var', 'Vif'])
    x_var_names = data.columns
    for i in range(0, x_var_names.shape[0]):
        y = data[x_var_names[i]]
        x = data[x_var_names.drop([x_var_names[i]])]
        r_squared = sm.OLS(y,x).fit().rsquared
        vif = round(1/(1-r_squared),2)
        vif_df.loc[i] = [x_var_names[i], vif]
    return vif_df.sort_values(by = 'Vif', axis = 0, ascending=False, inplace=False)


X=df.drop(['Salary'],axis=1)
calculate_vif(X)
```


Output:

	Var	Vif
2	2.0	9.94
0	0.0	9.84
1	1.0	7.80
3	3.0	1.23

From the output, we can see that the dummy variables which are created using one-hot encoding have VIF above 5. We have a multicollinearity problem.

Now, let us drop one of the dummy variables to solve the multicollinearity issue:

```
df = df.drop(df.columns[[0]], axis=1)
calculate_vif(df)
```

view rawone6.py hosted with by GitHub

Output:

	Var	Vif
2	3.0	2.60
1	2.0	1.91
0	1.0	1.69

Wow! VIF has decreased. We solved the problem of multicollinearity. Now, the dataset is ready for building the model.

We have seen two different techniques – Label and One-Hot Encoding for handling categorical variables. In the next section, I will touch upon when to prefer label encoding vs. One-Hot Encoding.

When to use a Label Encoding vs. One Hot Encoding

This question generally depends on your dataset and the model which you wish to apply. But still, a few points to note before choosing the right encoding technique for your model:

We apply One-Hot Encoding when:

1. The categorical feature is **not ordinal** (like the countries above)
2. The number of categorical features is less so one-hot encoding can be effectively applied

We apply Label Encoding when:

1. The categorical feature is **ordinal** (like Jr. kg, Sr. kg, Primary school, high school)
2. The number of categories is quite large as one-hot encoding can lead to high memory consumption