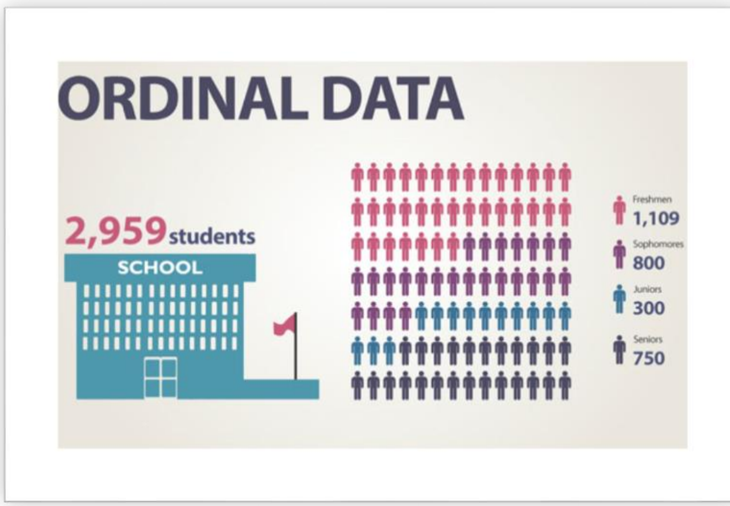
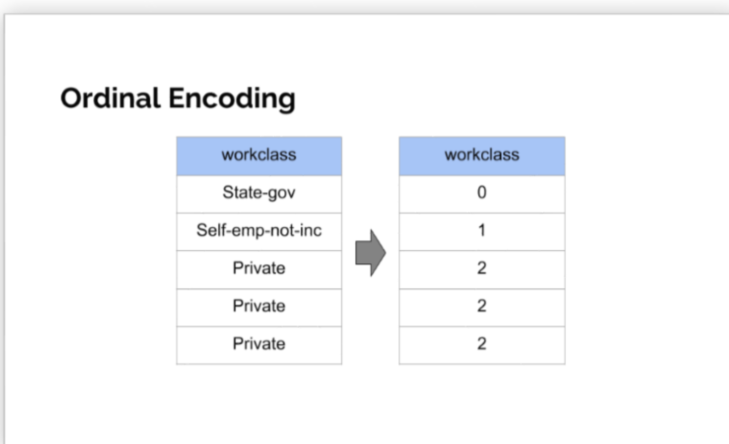
Feature Encoding with Python

ENCODING CATEGORICAL FEATURES FOR MACHINE LEARNING

Contributors ¯_(ツ)_/¯

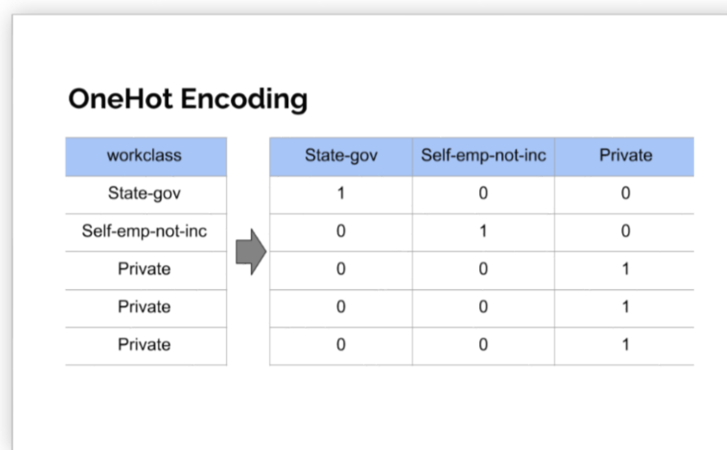
Vivek Chuadhary | Chintan Chitroda | Manvendra Singh





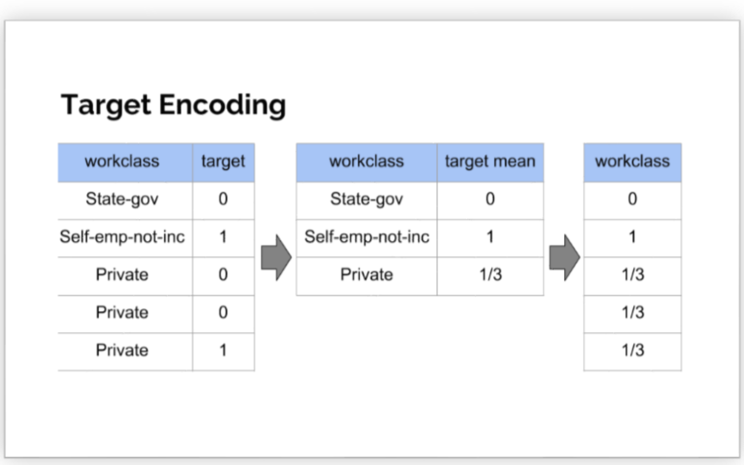
workclass
State-gov
Self-emp-not-inc
Private
Private
Private

workclass
0
1
2
2
2



workclass
State-gov
Self-emp-not-inc
Private
Private
Private

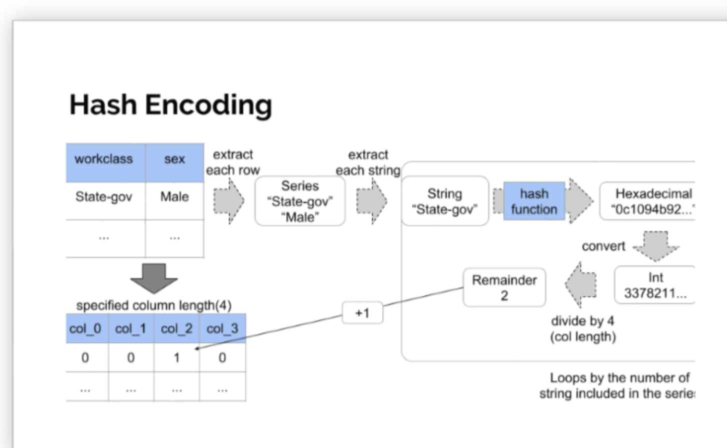
State-gov	Self-emp-not-inc	Private
1	0	0
0	1	0
0	0	1
0	0	1
0	0	1



workclass	target
State-gov	0
Self-emp-not-inc	1
Private	0
Private	0
Private	1

workclass	target mean
State-gov	0
Self-emp-not-inc	1
Private	1/3

workclass
0
1
1/3
1/3
1/3



Techniques we will be seeing:

- One Hot Encoding
- Frequency One Hot Encoding
- Label Encoding
- Mapping
- Feature Factorization
- Target Mean Encoding
- Frequency Encoding
- Binary Encoding
- Feature Hashing
- Comparing Encoding Technique

Download Data sets from here: https://drive.google.com/drive/folders/12WRchWpMhYTjofiH6rI0jAhjhva8--a1?usp=sharing

Encoding Methods

What is Encoding ? and use of it in DataScience/ Machine Learning ?

- In DataScience, We have many types of datatypes like String,int,Datetime etc. We know Computers & our Machine Learning Model can only understand and interpret Numeric data. Then What about String Data, how to deal with this types of String (categorical) data ? - This is where Encoding comes into the picture, Encoding is basically converting and representing the String data in a numeric format or represent them in such a way that our model is able to interpret that data.

This Book Contains 2 part::

> ```
> 1. Basics Encoding usage with Syntax/Example
> 2. Detailed Encoding usage with Syntax/Example on Realworld Dataset
> ```

Basic Encoding

There are many ways of Encoding Data, But here are some popular and widely used Encoding methods

1. OneHot Encoding:
    - one column for each value to compare vs. all other values. using 1s

- Frequency OneHot Encode:
    - one column for each value with High Frequency to compare vs. all other values. using 1s.
- Label Encoding:
    - convert string labels to integer values 1 through k.
- Frequency/Count Encoding:
    - Values encoded with frequency/count of it in column
- Target Mean Encoding:
    - uses the mean of the DV, must take steps to avoid overfitting/ response leakage. Nominal, ordinal. For classification tasks.
- Binary Encoding:
    - Uses label Encoding and then converts Integer to binary format creating column for each 0/1 value of binary

Import Pandas and Seaborn

In [2]:
```python
import pandas as pd
import seaborn as sns
```

1. ONE HOT ENCODING

In [3]:
```python
data = sns.load_dataset('tips') ## laoding dataset
```

In [4]:
```python
data.head(5)
```

	total_bill	tip	sex	smoker	day	time	size
0	16.99	1.01	Female	No	Sun	Dinner	2
1	10.34	1.66	Male	No	Sun	Dinner	3
2	21.01	3.50	Male	No	Sun	Dinner	3
3	23.68	3.31	Male	No	Sun	Dinner	2
4	24.59	3.61	Female	No	Sun	Dinner	4

In [5]:
```python
## Encoding Day column
data.day.value_counts()
```

```
Sat     87
Sun     76
Thur    62
Fri     19
Name: day, dtype: int64
```

In [9]:
```
#### Below we make seprate column for each value in column
pd.get_dummies(data.day).head(5)
```

day	Thur	Fri	Sat	Sun
0	0	0	0	1
1	0	0	0	1
2	0	0	0	1
3	0	0	0	1
4	0	0	0	1

In [6]:
```
#### Below we make seprate column for each value in column and drop 1st column
### As we have rest 3 columns with 1 and 0 we know that 4 column would be 0 or 1 so we drop it
pd.get_dummies(data.day,drop_first=True)
```

day	Fri	Sat	Sun
0	0	0	1
1	0	0	1
2	0	0	1
3	0	0	1
4	0	0	1
...	...	...	...
239	0	1	0
240	0	1	0
241	0	1	0
242	0	1	0
243	0	0	0

244 rows × 3 columns

2. Frequency OneHOT ENCODE

In [7]:
```
data = sns.load_dataset('planets')
```

In [8]:
```
data.method.value_counts()
```

```
Radial Velocity                  553
Transit                          397
Imaging                           38
Microlensing                      23
Eclipse Timing Variations          9
Pulsar Timing                      5
Transit Timing Variations          4
Orbital Brightness Modulation      3
Astrometry                         2
Pulsation Timing Variations        1
Name: method, dtype: int64
```

Think of it like we have too many methods so we won't make column for each value but only for top 5-10 values having most frequency in this case it's - First five from above table

In [9]:
```
temp = ['Radial Velocity',
'Transit',
'Imaging',
'Microlensing',
'Eclipse Timing Variations']
```

In [10]:
```
data.method = data.method.apply(lambda x: x if x in temp else 'unknown_method')
```

In [11]:
```
data.method.value_counts()
```

```
Radial Velocity              553
Transit                      397
Imaging                       38
Microlensing                  23
unknown_method                15
Eclipse Timing Variations      9
Name: method, dtype: int64
```

```
In [12]: pd.get_dummies(data.method)
```

	Eclipse Timing Variations	Imaging	Microlensing	Radial Velocity	Transit	unknown_method
0	0	0	0	1	0	0
1	0	0	0	1	0	0
2	0	0	0	1	0	0
3	0	0	0	1	0	0
4	0	0	0	1	0	0
...	...	...	...	...	...	...
1030	0	0	0	0	1	0
1031	0	0	0	0	1	0
1032	0	0	0	0	1	0
1033	0	0	0	0	1	0
1034	0	0	0	0	1	0

1035 rows × 6 columns

This method is Preferred when the column has too many unique values in it.

3. Label Encoding

In this we just replace values with some number

```
In [13]: data = sns.load_dataset('planets')
```

```
In [14]: data.method.value_counts()
```

```
Radial Velocity                  553
Transit                          397
Imaging                           38
Microlensing                      23
Eclipse Timing Variations          9
Pulsar Timing                      5
Transit Timing Variations          4
Orbital Brightness Modulation      3
Astrometry                         2
Pulsation Timing Variations        1
Name: method, dtype: int64
```

```
In [15]: from sklearn.preprocessing import LabelEncoder
```

```
In [16]: encoded_method = LabelEncoder.fit_transform(LabelEncoder,data.method)
```

```
In [17]: pd.DataFrame(data.method.value_counts().index, pd.Series(encoded_method).value_counts().index)
```

	0
7	Radial Velocity
8	Transit
2	Imaging
3	Microlensing
1	Eclipse Timing Variations
5	Pulsar Timing
9	Transit Timing Variations
4	Orbital Brightness Modulation
0	Astrometry
6	Pulsation Timing Variations

Above Here,we can see the Numer assigned to each of value

4. Frequency/Count Encoding:

```
In [18]: data = sns.load_dataset('planets')
```

```
In [19]: temp = dict(data.method.value_counts())
```

here we would be basically replacing the string with the frequency count it is present in column

In [20]:
```
temp ## so these are the numbers we would be replacing them with
```

```
{'Radial Velocity': 553,
 'Transit': 397,
 'Imaging': 38,
 'Microlensing': 23,
 'Eclipse Timing Variations': 9,
 'Pulsar Timing': 5,
 'Transit Timing Variations': 4,
 'Orbital Brightness Modulation': 3,
 'Astrometry': 2,
 'Pulsation Timing Variations': 1}
```

In [21]:
```
## so this is how we do it
freq_encode = [] # create list
for i in data.method:
    if i in temp.keys(): # iterate over keys
        freq_encode.append(temp[i]) # append value for that key
```

In [22]:
```
data.method = freq_encode # finally replacing list with column
```

In [23]:
```
data.method.value_counts()
```

```
553    553
397    397
38      38
23      23
9        9
5        5
4        4
3        3
2        2
1        1
Name: method, dtype: int64
```

in this if your values are too high just equalize them with standard scalaer or just divide the data with highest value in column

In [24]:
```
data.method/553
```

```
0       1.000000
1       1.000000
2       1.000000
3       1.000000
4       1.000000
          ...
1030    0.717902
1031    0.717902
1032    0.717902
1033    0.717902
1034    0.717902
Name: method, Length: 1035, dtype: float64
```

5. Target Mean Encoding

In [25]:
```
data = sns.load_dataset('planets')
```

In [26]:
```
data.head(5)
```

	method	number	orbital_period	mass	distance	year
0	Radial Velocity	1	269.300	7.10	77.40	2006
1	Radial Velocity	1	874.774	2.21	56.95	2008
2	Radial Velocity	1	763.000	2.60	19.84	2011
3	Radial Velocity	1	326.030	19.40	110.62	2007
4	Radial Velocity	1	516.220	10.50	119.47	2009

Think here like orbital_period is my target variable them i would just take the mean of my column wrt to my target varaible and the value (mean) would be encoded.

In [27]:
```
mean_encode = dict(data.groupby('method')['orbital_period'].mean())
```

In [28]:
```
mean_encode # this would be the values we would be replacing it with
```

```
{'Astrometry': 631.1800000000001,
 'Eclipse Timing Variations': 4751.644444444445,
 'Imaging': 118247.7375,
 'Microlensing': 3153.5714285714284,
 'Orbital Brightness Modulation': 0.7093065833333334,
 'Pulsar Timing': 7343.021201258,
 'Pulsation Timing Variations': 1170.0,
 'Radial Velocity': 823.3546800171247,
 'Transit': 21.102072671259457,
 'Transit Timing Variations': 79.7835}
```

In [29]:
```
## so this is how we do it
Tmean_encode = [] # create list
for i in data.method:
    if i in mean_encode.keys(): # iterate over keys
        Tmean_encode.append(mean_encode[i]) # append value for that key
```

In [30]:
```
data.method = Tmean_encode # finally replacing list with column
```

In [31]:
```
data.method.value_counts()
```

```
823.354680       553
21.102073        397
118247.737500     38
3153.571429       23
4751.644444        9
7343.021201        5
79.783500          4
0.709307           3
631.180000         2
1170.000000        1
Name: method, dtype: int64
```

you can scale if the values are too high with StandardScaler or MinMax or manullay divide by column by largest value in it.

In [32]:
```
from sklearn.preprocessing import StandardScaler
```

In [33]:
```
ss = StandardScaler()
```

In [34]:
```
ss.fit(data.method.values.reshape(-1,1))
```

```
StandardScaler(copy=True, with_mean=True, with_std=True)
```

In [35]:
```
data['Method_StandardScaled'] = ss.transform(data.method.values.reshape(-1,1))
```

In [36]:
```
data.head()
```

	method	number	orbital_period	mass	distance	year	Method_StandardScaled
0	823.35468	1	269.300	7.10	77.40	2006	-0.185923
1	823.35468	1	874.774	2.21	56.95	2008	-0.185923
2	823.35468	1	763.000	2.60	19.84	2011	-0.185923
3	823.35468	1	326.030	19.40	110.62	2007	-0.185923
4	823.35468	1	516.220	10.50	119.47	2009	-0.185923

Binary Encoding

Binary Encoding just labels values to integer then takes binary of the integer and makes binary table to encode data
Binary,Interger

- 0000 - 0
- 0001 - 1
- 0010 - 2
- 0011 - 3
- 0100 - 4
- 0101 - 5
- 0110 - 6
- 0111 - 7
- 1000 - 8
- 1001 - 9
- 1010 - 10

We will we using category_encoders Python Package for this.

- category_encoders can be used to encode all the above mentioned encoding techniques as well.

In [91]:
```
import category_encoders as ce # include Category Encoders Package
```

In [92]:
```
# Create dataframe with basic city names
data = pd.DataFrame({
    'city' : ['delhi','mumbai','pune','chandigarh','nasik', 'hyderabad',
              'lukhnow', 'gurgaon', 'odisa','bangluru','mumbai']
})
```

In [93]:
```
# create an object of the OrdinalEncoding
be = ce.BinaryEncoder()
# fit and transform and you will get the encoded data and store in temp dataframe
temp = be.fit_transform(data)
```

In [94]:
```
temp ## our encoded binary dataframe
```

	city_0	city_1	city_2	city_3	city_4
0	0	0	0	0	1
1	0	0	0	1	0
2	0	0	0	1	1
3	0	0	1	0	0
4	0	0	1	0	1
5	0	0	1	1	0
6	0	0	1	1	1
7	0	1	0	0	0
8	0	1	0	0	1
9	0	1	0	1	0
10	0	0	0	1	0

In [95]:
```
temp['city'] = data.city ## adding city column
```

In [96]:
```
## Mapping Integer Manully to show how it works
temp['integer_num'] = temp['city'].map({'delhi':1,'mumbai':2,'pune':3,'chandigarh':4,'nasik':5, 'h
yderabad':6,
                        'lukhnow':7, 'gurgaon':8, 'odisa':9,'bangluru':10})
```

In [97]:
```
temp
```

	city_0	city_1	city_2	city_3	city_4	city	integer_num
0	0	0	0	0	1	delhi	1
1	0	0	0	1	0	mumbai	2
2	0	0	0	1	1	pune	3
3	0	0	1	0	0	chandigarh	4
4	0	0	1	0	1	nasik	5
5	0	0	1	1	0	hyderabad	6
6	0	0	1	1	1	lukhnow	7
7	0	1	0	0	0	gurgaon	8
8	0	1	0	0	1	odisa	9
9	0	1	0	1	0	bangluru	10
10	0	0	0	1	0	mumbai	2

here we can see the city delhi was encoded as INTERGER '1' AND THEN BINARY WAS FORMED '00001' Mumbai was labeled integer '2' and then binarized '00010' and same for rest. The no of column created is dependent on the highest values of integer

Thank You

This was the syntax and Basic Part

Now Lets how we use this all and more encoding Techniques in Real world and Huge Dataset

Detailed Explaination - Feature Encoding

Generally in our dataset we have 2 types of features

1. Numerical (Integer,floats)
2. Categorical (Nominal, ordinal)

We cannot pass in categorical features in Machine Learning models. So we need to convert them into numeric features.

Categorical Variables are of 2 types Ordinal and Nominal.

- Ordinal variables has some kind order. (Good, Better, Best), (First, Second, Third)
- Nominal variables has no ordering between them. (Cat, Dog, Monkey), (Apple, Banana, Mango)

Based on categorical variables whether they are ordinal or nominal we appply different techniques on them.

In [0]:
```
#Let's create a dataframe
import pandas as pd
df = pd.DataFrame ({'country' : ['India','U.S','Australia','India','Australia','India','U.S'],
                    'Age' : [44,34,28,27,30,42,25],
                    'Salary' : [72000,44000,35000,27000,32000,56000,45000],
                    'Purchased' : ['yes','no','yes','yes','no','yes','no']
                    })
```

In [0]:
```
#Let's check our dataframe
print(df)
```

```
     country  Age  Salary Purchased
0      India   44   72000       yes
1        U.S   34   44000        no
2  Australia   28   35000       yes
3      India   27   27000       yes
4  Australia   30   32000        no
5      India   42   56000       yes
6        U.S   25   45000        no
```

In [0]:
```
#check the datatypes
df.dtypes
```

```
country      object
Age           int64
Salary        int64
Purchased    object
dtype: object
```

Here we have 2 categorical feature

- Country.
- Purchased.

Age and Salary have numeric values.

We know it well that we cannot pass in categorical values in our models.

Label Encoding

In [0]:
```
df['country'].unique() #check unique
```

```
array(['India', 'U.S', 'Australia'], dtype=object)
```

So Here we have 3 categories in country column.

- India
- U.S
- Australia

In label encoding different categories are given different unique values starting from 0 to (n-1). n is the number of categories.

In [0]:
```python
from sklearn.preprocessing import LabelEncoder #import the LabelEncoder from sklrean library
le= LabelEncoder()     #create the instance of LabelEncoder

df['country_temp'] = le.fit_transform(df['country'])    #apply LabelEncoding of country column
```

In [0]:
```python
df['country_temp']
```

```
0    1
1    2
2    0
3    1
4    0
5    1
6    2
Name: country_temp, dtype: int64
```

Here we can see that country feature has been tranformed into numeric values. Label encoding is done in alphabatical order as we can see here.

- Australia -----> 0
- India --------> 1
- U.S ---------> 2

Problem With Label Encoding

Here we have assigned numeric values i.e (0-Australia), (1-India), (2-U.S) in the same column. Problem here is that the machine learning models won't interpret these values as different labels as 0 < 1 < 2. Our model might interpret them in some order. But we don't have any ordering in our country feature. we cannot say Australia < India < U.S .

We use One Hot encoding to overcome this problem. It is also known as nominal encoding. Here We create 3 different columns [India, Australia, U.S]. We assign 1 if that label is present in particular row otherwise we marks it as 0.

In [0]:
```python
#we will use get_dummies to do One Hot encoding
pd.get_dummies(df['country'])
```

	Australia	India	U.S
0	0	1	0
1	0	0	1
2	1	0	0
3	0	1	0
4	1	0	0
5	0	1	0
6	0	0	1

- Here in first row ['India'] is assigned 1 and Australia and U.S are assigned 0.
- Similarly in 2nd row ['U.S'] is assigned 1 and other columns are assigned 0.

We can drop the first column here, it is just increasing the features. Reason ---- Even if we just have two columns suppose india and U.S and both are assigned 0. It is understood that when both of these labels are zero The 3rd label is automatically going to be 1.

In [0]:
```python
#Dropping the first column
pd.get_dummies(df['country'],drop_first=True)
```

	India	U.S
0	1	0
1	0	1
2	0	0
3	1	0
4	0	0
5	1	0
6	0	1

Here we have done one hot encoding only on single feature but in real world datasets there will be many categorical features. Suppose our dataset has 50 categorical features with 3 different labels in each features. In that case if we apply one hot encoding, our features will also increase. we will have 100 features. It will make our model more complex.

Based on the dataset there are different techniques that we can apply to over-come this problem of dimensionality.

Binary Encoding

This is not intiuative like the previous ones. Here the labels are firstly encoded ordinal and then they are converted into binary codes. Then the digits from that binary string are converted into different features.

In [0]:
```
#create 1 more column occupation here
df['occupation'] = ['Self-employeed','Freelancer','Family-business','Data-scientist','Pensioner',
'Manager','Daily-wage-worker']
print(df['occupation'])
```

```
0       Self-employeed
1           Freelancer
2      Family-business
3       Data-scientist
4            Pensioner
5              Manager
6    Daily-wage-worker
Name: occupation, dtype: object
```

We have seven different categories here. And we don't have any ordering in them as well.

In [0]:
```
#install category_encoders first
!pip install category_encoders
```

```
Collecting category_encoders
  Downloading https://files.pythonhosted.org/packages/a0/52/c54191ad3782de633ea3d6ee3bb2837bda0cf3bc97644bb6375cf14150a0/category_encoders
-2.1.0-py2.py3-none-any.whl (100kB)
     |████████████████████████████████| 102kB 2.1MB/s
Requirement already satisfied: statsmodels>=0.6.1 in /usr/local/lib/python3.6/dist-packages (from category_encoders) (0.10.2)
Requirement already satisfied: pandas>=0.21.1 in /usr/local/lib/python3.6/dist-packages (from category_encoders) (1.0.3)
Requirement already satisfied: scipy>=0.19.0 in /usr/local/lib/python3.6/dist-packages (from category_encoders) (1.4.1)
Requirement already satisfied: scikit-learn>=0.20.0 in /usr/local/lib/python3.6/dist-packages (from category_encoders) (0.22.2.post1)
Requirement already satisfied: patsy>=0.4.1 in /usr/local/lib/python3.6/dist-packages (from category_encoders) (0.5.1)
Requirement already satisfied: numpy>=1.11.3 in /usr/local/lib/python3.6/dist-packages (from category_encoders) (1.18.2)
Requirement already satisfied: pytz>=2017.2 in /usr/local/lib/python3.6/dist-packages (from pandas>=0.21.1->category_encoders) (2018.9)
Requirement already satisfied: python-dateutil>=2.6.1 in /usr/local/lib/python3.6/dist-packages (from pandas>=0.21.1->category_encoders)
(2.8.1)
Requirement already satisfied: joblib>=0.11 in /usr/local/lib/python3.6/dist-packages (from scikit-learn>=0.20.0->category_encoders) (0.1
4.1)
Requirement already satisfied: six in /usr/local/lib/python3.6/dist-packages (from patsy>=0.4.1->category_encoders) (1.12.0)
Installing collected packages: category-encoders
Successfully installed category-encoders-2.1.0
```

In [0]:
```
# we will use BinaryEncoder from category_encoders library to do binary encoding
import category_encoders as ce
encoder = ce.BinaryEncoder(cols = ['occupation'])
df_binary = encoder.fit_transform(df)
print(df_binary)
```

```
     country  Age  Salary  ... occupation_1  occupation_2  occupation_3
0      India   44   72000  ...            0             0             1
1        U.S   34   44000  ...            0             1             0
2  Australia   28   35000  ...            0             1             1
3      India   27   27000  ...            1             0             0
4  Australia   30   32000  ...            1             0             1
5      India   42   56000  ...            1             1             0
6        U.S   25   45000  ...            1             1             1

[7 rows x 9 columns]
```

We had 7 different categories in occupation if we would have used one hot encoding it would have given us 7 features. But by using Binary Encoding we have limited it to 3. Binary Encoding is very useful when we have many categories within a single feature. It help us to reduce the dimensionality.

In [0]:
```
'''we have seen 3 basic types feature encoding techniques here there are many more.
               we will look at them with some practical uses and with some real world dataset'''
```

Lets Try Another Dataset

We are going to apply the different encoding techniques on big mart sales data.

Things to learn -

- Indentifying data type as ordinal,nominal and continuous.
- Applying different types of encoding.
- Challenges with different encoding techniques.
- Choosing the appropriate encoding techniques.

In [0]:

```
import pandas as pd #import pandas
import numpy as np #import numpy
from sklearn.preprocessing import LabelEncoder  #importing LabelEncoder
import warnings
warnings.filterwarnings("ignore")
```

In [0]:

```
train = pd.read_csv('/content/drive/My Drive/Feature Encoding/feature_en/Feature_encoding/train_bm.csv')
```

In [0]:

```
#check the head of dataset
train.head(5)
```

	Item_Identifier	Item_Weight	Item_Fat_Content	Item_Visibility	Item_Type	Item_MRP	Outlet_Identifier	Outlet_Establishment_Ye
0	FDA15	9.30	Low Fat	0.016047	Dairy	249.8092	OUT049	1999
1	DRC01	5.92	Regular	0.019278	Soft Drinks	48.2692	OUT018	2009
2	FDN15	17.50	Low Fat	0.016760	Meat	141.6180	OUT049	1999
3	FDX07	19.20	Regular	0.000000	Fruits and Vegetables	182.0950	OUT010	1998
4	NCD19	8.93	Low Fat	0.000000	Household	53.8614	OUT013	1987

In [0]:

```
#check the size of the dataset
print('Data has {} Number of rows'.format(train.shape[0]))
print('Data has {} Number of columns'.format(train.shape[1]))
```

```
Data has 8523 Number of rows
Data has 12 Number of columns
```

In [0]:

```
#check the information of the dataset
train.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 8523 entries, 0 to 8522
Data columns (total 12 columns):
 #   Column                     Non-Null Count  Dtype
---  ------                     --------------  -----
 0   Item_Identifier            8523 non-null   object
 1   Item_Weight                7060 non-null   float64
 2   Item_Fat_Content           8523 non-null   object
 3   Item_Visibility            8523 non-null   float64
 4   Item_Type                  8523 non-null   object
 5   Item_MRP                   8523 non-null   float64
 6   Outlet_Identifier          8523 non-null   object
 7   Outlet_Establishment_Year  8523 non-null   int64
 8   Outlet_Size                6113 non-null   object
 9   Outlet_Location_Type       8523 non-null   object
 10  Outlet_Type                8523 non-null   object
 11  Item_Outlet_Sales          8523 non-null   float64
dtypes: float64(4), int64(1), object(7)
memory usage: 799.2+ KB
```

As we can see here, we have 7 categorical variables and 5 numeric variables. The first task is to identify these categorical variables as nominal or ordinal.

In [0]:

```
#let's keep our categorical variables in one table
cat_data = train[['Item_Identifier','Item_Fat_Content','Item_Type','Outlet_Identifier','Outlet_Size','Outle
```

In [0]:

```
cat_data.head()   #check the head of categorical data
```

	Item_Identifier	Item_Fat_Content	Item_Type	Outlet_Identifier	Outlet_Size	Outlet_Location_Type	Outlet_Type
0	FDA15	Low Fat	Dairy	OUT049	Medium	Tier 1	Supermarket Type1
1	DRC01	Regular	Soft Drinks	OUT018	Medium	Tier 3	Supermarket Type2
2	FDN15	Low Fat	Meat	OUT049	Medium	Tier 1	Supermarket Type1
3	FDX07	Regular	Fruits and Vegetables	OUT010	NaN	Tier 3	Grocery Store
4	NCD19	Low Fat	Household	OUT013	High	Tier 3	Supermarket Type1

In [0]:

```
cat_data.apply(lambda x: x.unique()) #check the number of unique values in each column
```

```
Item_Identifier         1559
Item_Fat_Content           5
Item_Type                 16
Outlet_Identifier         10
Outlet_Size                3
Outlet_Location_Type       3
Outlet_Type                4
dtype: int64
```

Now think which encoding technique can we apply here.

- First thought would be to apply one hot encoding on features which has 3-5 unique categories.
- But what if there is some kind of ordering present between them. So firstly we should identify the nominal and ordinal variable
- Let's check one by one

In [0]:

```
#check the top 10 frequency in Item_Identifier
cat_data['Item_Identifier'].value_counts().head(10)
```

```
FDG33    10
FDW13    10
FDP25     9
FDQ40     9
FDX20     9
FDW49     9
FDX31     9
NCF42     9
FDT07     9
DRN47     9
Name: Item_Identifier, dtype: int64
```

The values in Item_Identifier has no ordering as we can see. These are nominal categorical variable.

The first column has 1559 unique values. If we try to do one hot encoding here we will have 1558 new features. We cannot feed in these many features in our model. It will make our model complex and it will reduce the model accuracy.

In [0]:

```
pd.get_dummies(cat_data['Item_Identifier'],drop_first=True)  #applying one hot encoding
```

	DRA24	DRA59	DRB01	DRB13	DRB24	DRB25	DRB48	DRC01	DRC12	DRC13	DRC24	DRC25	DRC27	DRC36	DRC49
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
8518	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
8519	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
8520	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
8521	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
8522	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

8523 rows × 1558 columns

As expected from a single feature now we have 1558 features. So it's a bad idea to apply one hot encoding here. We should not apply one hot encoding when there are too many categories.

So one hot encoding has failed us here. Now for rescue we move to LabelEncoding but we are very much aware that if we apply label encoding on a feature it assigns a natural ranking to the categories alphabatically. So we cannot apply Label encoding as well.

So we have 1 thing left (Binary Encoding) that we have learnt previously. Let's apply it and see what we get.

In [0]:

```
#apply binary encoding on Item_Identifier
import category_encoders as ce                          #import category_encoders
encoder = ce.BinaryEncoder(cols=['Item_Identifier'])    #create instance of binary enocder
df_binary = encoder.fit_transform(cat_data)             #fit and tranform on cat_data
df_binary.head(5)
```

	Item_Identifier_0	Item_Identifier_1	Item_Identifier_2	Item_Identifier_3	Item_Identifier_4	Item_Identifier_5	Item_Identifier_6	Ite
0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0

In [0]:

```
#check the columns
df_binary.columns
```

```
Index(['Item_Identifier_0', 'Item_Identifier_1', 'Item_Identifier_2',
       'Item_Identifier_3', 'Item_Identifier_4', 'Item_Identifier_5',
       'Item_Identifier_6', 'Item_Identifier_7', 'Item_Identifier_8',
       'Item_Identifier_9', 'Item_Identifier_10', 'Item_Identifier_11',
       'Item_Fat_Content', 'Item_Type', 'Outlet_Identifier', 'Outlet_Size',
       'Outlet_Location_Type', 'Outlet_Type'],
      dtype='object')
```

Binary encoder has given us 11 new feature which is way less than we were getting from one hot encoding. So we have been rescued here by Binary Encoding.

We have applied binary encoding but it doesn't provide us any intution as how these new features are made. All we know is by using binary encoding Here the labels are firstly encoded ordinal and then they are converted into binary codes. Then the digits from that binary string are converted into different features.

There are other intutive measures to reduce the features. We will look at them later.

##Encoding Item_Fat_Content

In [0]:
```
#check the unique values
cat_data['Item_Fat_Content'].unique()
```

```
array(['Low Fat', 'Regular', 'low fat', 'LF', 'reg'], dtype=object)
```

Here we have 5 unique values but if we look at them closely there are only 2 unique values. Low Fat and Regular, others are just short forms for them or are in small letters.

In [0]:
```
low_fat = ['LF','low fat']
cat_data['Item_Fat_Content'].replace(low_fat,'Low Fat',inplace = True) #replace 'LF' and 'low fat' with 'Lo
cat_data['Item_Fat_Content'].replace('reg','Regular',inplace = True)   #Replace 'reg' with regular
```

In [0]:
```
cat_data['Item_Fat_Content'].unique()
```

```
array(['Low Fat', 'Regular'], dtype=object)
```

Here we have 2 categories in Item_Fat_Content and we have some ordering between the. Low Fat will have less Fat content than the regular Fat. So it is a ordinal variable.

In [0]:
```
#Apply LabelEncoder
le = LabelEncoder()
cat_data['Item_Fat_Content_temp'] = le.fit_transform(cat_data['Item_Fat_Content'])
print(cat_data['Item_Fat_Content'].head())
print(cat_data['Item_Fat_Content_temp'].head())
```

```
0    Low Fat
1    Regular
2    Low Fat
3    Regular
4    Low Fat
Name: Item_Fat_Content, dtype: object
0    0
1    1
2    0
3    1
4    0
Name: Item_Fat_Content_temp, dtype: int64
```

Here we only had 2 categories 'Low Fat' and 'Regular' so using LabelEncoding has worked here. It has mapped :-

- Low Fat ------- 0
- Regular ------- 1

Here the natural ranking of alphabets has worked but every time you are not this lucky.

##We can use map to do ordinal encoding

In [0]:
```
#prepare a dict to map
mapping = {'Low Fat' : 0,'Regular': 1} #map Low Fat as 0 and Regular as 1
cat_data['Item_Fat_Content_temp1'] = cat_data['Item_Fat_Content'].map(mapping)
cat_data['Item_Fat_Content_temp1'].head()
```

```
0    0
1    1
2    0
3    1
4    0
Name: Item_Fat_Content_temp1, dtype: int64
```

It is useful when we have ordering in our categories.

Use Pandas pd.factorize method.

It does the nominal encoding based on the order in which the categories apper. If Low Fat is at index 0 then it will be encoded as 0 Regular as 1 and vice versa.

In [0]:

```python
factorized,index = pd.factorize(cat_data['Item_Fat_Content'])  #using pd.factorize it gives us factorized a
print(factorized)
print(index)
```

```
[0 1 0 ... 0 1 0]
Index(['Low Fat', 'Regular'], dtype='object')
```

In this Notebook we have seen 2 new encoding techniques.

- Mapping
- pd.factorize

We have seen the usage of different methods, their advantages and disadvantages.

In [0]:

```python
#Let's look at item type column
print(cat_data['Item_Type'].nunique())  #check number of unique values
print(cat_data['Item_Type'].unique())   #check the unique values
```

```
16
['Dairy' 'Soft Drinks' 'Meat' 'Fruits and Vegetables' 'Household'
 'Baking Goods' 'Snack Foods' 'Frozen Foods' 'Breakfast'
 'Health and Hygiene' 'Hard Drinks' 'Canned' 'Breads' 'Starchy Foods'
 'Others' 'Seafood']
```

And we don't Have any ordering between them. So we have to apply ordinal encoding technique. i Leave it upto you to decide which technique to apply and we will have look at other techniques in our next Notebook.

Till now we have looked at 6 feature encoding techniques.

- Label Encoding
- One Hot Encoding
- Binary Encoding
- Mapping
- pd.factorize

In this notebook we will look at 2 new encoding techniques.

- Frequency Encoding
- Mean Encoding

In [0]:
```
import pandas as pd #import pandas
import numpy as np #import numpy
from sklearn.preprocessing import LabelEncoder  #importing LabelEncoder
import warnings
warnings.filterwarnings("ignore")
```

In [0]:
```
train = pd.read_csv('/content/drive/My Drive/Feature Encoding/feature_en/Feature_encoding/train_b
m.csv')
```

In [0]:
```
train.head()
```

	Item_Identifier	Item_Weight	Item_Fat_Content	Item_Visibility	Item_Type	Item_MRP	Outlet_Identifier	Outlet_Establishment
0	FDA15	9.30	Low Fat	0.016047	Dairy	249.8092	OUT049	1999
1	DRC01	5.92	Regular	0.019278	Soft Drinks	48.2692	OUT018	2009
2	FDN15	17.50	Low Fat	0.016760	Meat	141.6180	OUT049	1999
3	FDX07	19.20	Regular	0.000000	Fruits and Vegetables	182.0950	OUT010	1998
4	NCD19	8.93	Low Fat	0.000000	Household	53.8614	OUT013	1987

In [0]:
```
#check the size of the dataset
print('Data has {} Number of rows'.format(train.shape[0]))
print('Data has {} Number of columns'.format(train.shape[1]))
```

```
Data has 8523 Number of rows
Data has 12 Number of columns
```

In [0]:
```
#let's keep our categorical variables in one table
cat_data = train[['Item_Identifier','Item_Fat_Content','Item_Type','Outlet_Identifier','Outlet_Siz
e','Outlet_Location_Type','Outlet_Type','Item_Outlet_Sales']]
```

In [0]:
```
cat_data.head()    #check the head of categorical data
```

	Item_Identifier	Item_Fat_Content	Item_Type	Outlet_Identifier	Outlet_Size	Outlet_Location_Type	Outlet_Type	Item_Outlet_
0	FDA15	Low Fat	Dairy	OUT049	Medium	Tier 1	Supermarket Type1	3735.1380
1	DRC01	Regular	Soft Drinks	OUT018	Medium	Tier 3	Supermarket Type2	443.4228
2	FDN15	Low Fat	Meat	OUT049	Medium	Tier 1	Supermarket Type1	2097.2700
3	FDX07	Regular	Fruits and Vegetables	OUT010	NaN	Tier 3	Grocery Store	732.3800
4	NCD19	Low Fat	Household	OUT013	High	Tier 3	Supermarket Type1	994.7052

In [0]:
```
#Let's start where we had left
print(cat_data['Item_Type'].nunique())
print(cat_data['Item_Type'].unique())
```

```
16
['Dairy' 'Soft Drinks' 'Meat' 'Fruits and Vegetables' 'Household'
 'Baking Goods' 'Snack Foods' 'Frozen Foods' 'Breakfast'
 'Health and Hygiene' 'Hard Drinks' 'Canned' 'Breads' 'Starchy Foods'
 'Others' 'Seafood']
```

Here we have 16 unique labels. And there is no ordering so it is a nominal category.

Frequency Encoding

It is a way to utilize the frequency of labels.

In [0]:
```
fe = cat_data['Item_Type'].value_counts(ascending=True)/len(cat_data)  #count the frequency of lab
els
print(fe)
```

```
Seafood                  0.007509
Breakfast                0.012906
Starchy Foods            0.017365
Others                   0.019829
Hard Drinks              0.025109
Breads                   0.029450
Meat                     0.049865
Soft Drinks              0.052212
Health and Hygiene       0.061011
Baking Goods             0.076030
Canned                   0.076147
Dairy                    0.080019
Frozen Foods             0.100434
Household                0.106770
Snack Foods              0.140795
Fruits and Vegetables    0.144550
Name: Item_Type, dtype: float64
```

In [0]:
```
cat_data['Item_Type'].map(fe).head(10)  #map frequency to item type
```

```
0    0.080019
1    0.052212
2    0.049865
3    0.144550
4    0.106770
5    0.076030
6    0.140795
7    0.140795
8    0.100434
9    0.100434
Name: Item_Type, dtype: float64
```

This technique is useful when the frequency is somewhat related with the target variable.

Mean Encoding

It is the most followed approach by the kagglers. We will not go into it's technality here. We will just look at it use and it's drawback.

We go through following steps for mean encoding

1. Group by categorical variable and obtain aggregated sum over target
2. Group by categorical variable and obtain aggregated count over target
3. divide step 2 / step 1

In [0]:
```
#get the mean of target variable label wise
me = cat_data.groupby('Outlet_Identifier')['Item_Outlet_Sales'].mean()
print(me)
```

```
Outlet_Identifier
OUT010     339.351662
OUT013    2298.995256
OUT017    2340.675263
OUT018    1995.498739
OUT019     340.329723
OUT027    3694.038558
OUT035    2438.841866
OUT045    2192.384798
OUT046    2277.844267
OUT049    2348.354635
Name: Item_Outlet_Sales, dtype: float64
```

In [0]:
```
#get the mean of target variable label wise
cat_data['Outlet_Identifier'].map(me).head(10)
```

```
0    2348.354635
1    1995.498739
2    2348.354635
3     339.351662
4    2298.995256
5    1995.498739
6    2298.995256
7    3694.038558
8    2192.384798
9    2340.675263
Name: Outlet_Identifier, dtype: float64
```

Here we have mapped different labels with the mean of the target variable.

When we have large number of features mean encoding is a way to go about encoding. As it doesnot creates any new feature. It also correlates with the target feature.

The disadvantage of mean encoding is that it is prone to overfitting.

In [0]:
```
#check value counts in Outlet_Size
cat_data['Outlet_Size'].value_counts()
```

```
Medium    2793
Small     2388
High       932
Name: Outlet_Size, dtype: int64
```

It is a ordinal variable we will make a dictionary as assign

- Small-----> 0
- Medium -----> 1
- High -----> 2

In [0]:
```
#Check the null values
cat_data['Outlet_Size'].isnull().sum()
```

```
2410
```

In [0]:
```
#fill the null values with other category for now
cat_data['Outlet_Size'].fillna('Others',inplace = True)
```

In [0]:
```
#prepare a dictionary to map
size_fe = {"Small" : 0, "Medium" : 1, "High" : 2, "Others" : 3}
cat_data['Outlet_Size'].map(size_fe).head(10)
```

```
0    1
1    1
2    1
3    3
4    2
5    1
6    2
7    1
8    3
9    3
Name: Outlet_Size, dtype: int64
```

In [0]:
```
cat_data['Outlet_Location_Type'].value_counts()
```

```
Tier 3    3350
Tier 2    2785
Tier 1    2388
Name: Outlet_Location_Type, dtype: int64
```

Here Tier 1, Teir 2 and Teir 3 are ordinal variables. We can use Label Encoding or map the values.

- Tier 3-----> 0
- Tier 2 -----> 1
- Tier 1-----> 2

```
In [0]: location_fe = {"Tier 3" : 1, "Tier 2" : 2, "Tier 1" : 3}
        cat_data['Outlet_Location_Type'].map(location_fe).head(10)
```

```
0    3
1    1
2    3
3    1
4    1
5    1
6    1
7    1
8    2
9    2
Name: Outlet_Location_Type, dtype: int64
```

```
In [0]: #Check last variable and do the encoding
        cat_data['Outlet_Type'].value_counts()
```

```
Supermarket Type1    5577
Grocery Store        1083
Supermarket Type3     935
Supermarket Type2     928
Name: Outlet_Type, dtype: int64
```

The labels here are nominal. It will be better to use nominal encoding. We have only 4 labels we can try one hot encoding or binary encoding as well.

```
In [0]: pd.get_dummies(cat_data['Outlet_Type'],drop_first=True).head()
```

	Supermarket Type1	Supermarket Type2	Supermarket Type3
0	1	0	0
1	0	1	0
2	1	0	0
3	0	0	0
4	1	0	0

Next we will use all the encoding techniques we have learnt till now on different datasets. So that you will have some practice and will have better understanding when to use which encoding.

Hash Encoding/Feature Hashing

Hash Encoding turns sequences of symbolic feature names (strings) into scipy.sparse matrices, using a hash function to compute the matrix column corresponding to a name. The hash function employed is the signed 32-bit version of Murmurhash3. Feature names of type byte string are used as-is. Unicode strings are converted to UTF-8 first, but no Unicode normalization is done. Feature values must be (finite) numbers. This class is a low-memory alternative to DictVectorizer and CountVectorizer, intended for large-scale (online) learning and situations where memory is tight, e.g. when running prediction code on embedded devices.

Big Advantage it can handle null values so no data cleaning is required

In [2]:
```
import pandas as pd
import numpy as np
```

In [3]:
```
df=pd.read_csv('hotel_bookings.csv')
```

In [4]:
```
df.head()
```

	hotel	is_canceled	lead_time	arrival_date_year	arrival_date_month	arrival_date_week_number	arrival_date_day_of_month
0	Resort Hotel	0	342	2015	July	27	1
1	Resort Hotel	0	737	2015	July	27	1
2	Resort Hotel	0	7	2015	July	27	1
3	Resort Hotel	0	13	2015	July	27	1
4	Resort Hotel	0	14	2015	July	27	1

5 rows × 32 columns

In [5]:
```
df.shape
```

```
(119390, 32)
```

In [6]:
```
df['hotel'].unique()
```

```
array(['Resort Hotel', 'City Hotel'], dtype=object)
```

map function replaces values with keys provided in dictionary format

In [7]:
```
df['hotel'] = df['hotel'].map({'Resort Hotel':0, 'City Hotel':1})
df['hotel'].unique()
```

```
array([0, 1], dtype=int64)
```

In [8]:
```
df['arrival_date_month'].unique()
```

```
array(['July', 'August', 'September', 'October', 'November', 'December',
       'January', 'February', 'March', 'April', 'May', 'June'],
      dtype=object)
```

In [9]:
```
df['arrival_date_month'] = df['arrival_date_month'].map({'January':1, 'February': 2, 'March':3, 'A
pril':4, 'May':5, 'June':6, 'July':7,
                                                                      'August':8, 'September':9, 'October':1
0, 'November':11, 'December':12})
df['arrival_date_month'].unique()
```

```
array([ 7,  8,  9, 10, 11, 12,  1,  2,  3,  4,  5,  6], dtype=int64)
```

In [10]:
```python
df.head()
```

	hotel	is_canceled	lead_time	arrival_date_year	arrival_date_month	arrival_date_week_number	arrival_date_day_of_month
0	0	0	342	2015	7	27	1
1	0	0	737	2015	7	27	1
2	0	0	7	2015	7	27	1
3	0	0	13	2015	7	27	1
4	0	0	14	2015	7	27	1

5 rows × 32 columns

In [11]:
```python
df['reservation_status'].unique()
```

```
array(['Check-Out', 'Canceled', 'No-Show'], dtype=object)
```

In [12]:
```python
df['customer_type'].unique()
```

```
array(['Transient', 'Contract', 'Transient-Party', 'Group'], dtype=object)
```

In [13]:
```python
df['meal'].unique()
```

```
array(['BB', 'FB', 'HB', 'SC', 'Undefined'], dtype=object)
```

In [14]:
```python
df['market_segment'].unique()
```

```
array(['Direct', 'Corporate', 'Online TA', 'Offline TA/TO',
       'Complementary', 'Groups', 'Undefined', 'Aviation'], dtype=object)
```

In [15]:
```python
df['distribution_channel'].unique()
```

```
array(['Direct', 'Corporate', 'TA/TO', 'Undefined', 'GDS'], dtype=object)
```

In [16]:
```python
df['reserved_room_type'].unique()
```

```
array(['C', 'A', 'D', 'E', 'G', 'F', 'H', 'L', 'P', 'B'], dtype=object)
```

In [17]:
```python
df['assigned_room_type'].unique()
```

```
array(['C', 'A', 'D', 'E', 'G', 'F', 'I', 'B', 'H', 'P', 'L', 'K'],
      dtype=object)
```

In [18]:
```python
df.columns
```

```
Index(['hotel', 'is_canceled', 'lead_time', 'arrival_date_year',
       'arrival_date_month', 'arrival_date_week_number',
       'arrival_date_day_of_month', 'stays_in_weekend_nights',
       'stays_in_week_nights', 'adults', 'children', 'babies', 'meal',
       'country', 'market_segment', 'distribution_channel',
       'is_repeated_guest', 'previous_cancellations',
       'previous_bookings_not_canceled', 'reserved_room_type',
       'assigned_room_type', 'booking_changes', 'deposit_type', 'agent',
       'company', 'days_in_waiting_list', 'customer_type', 'adr',
       'required_car_parking_spaces', 'total_of_special_requests',
       'reservation_status', 'reservation_status_date'],
      dtype='object')
```

In [19]:
```python
df['deposit_type'].unique()
```

```
array(['No Deposit', 'Refundable', 'Non Refund'], dtype=object)
```

In [20]:
```python
df.head(2)
```

	hotel	is_canceled	lead_time	arrival_date_year	arrival_date_month	arrival_date_week_number	arrival_date_day_of_month
0	0	0	342	2015	7	27	1
1	0	0	737	2015	7	27	1

2 rows × 32 columns

In [21]:
```
df.isnull().sum()
```

```
hotel                                0
is_canceled                          0
lead_time                            0
arrival_date_year                    0
arrival_date_month                   0
arrival_date_week_number             0
arrival_date_day_of_month            0
stays_in_weekend_nights              0
stays_in_week_nights                 0
adults                               0
children                             4
babies                               0
meal                                 0
country                            488
market_segment                       0
distribution_channel                 0
is_repeated_guest                    0
previous_cancellations               0
previous_bookings_not_canceled       0
reserved_room_type                   0
assigned_room_type                   0
booking_changes                      0
deposit_type                         0
agent                            16340
company                         112593
days_in_waiting_list                 0
customer_type                        0
adr                                  0
required_car_parking_spaces          0
total_of_special_requests            0
reservation_status                   0
reservation_status_date              0
dtype: int64
```

In [22]:
```
#Defining the independent variables and dependent variables
x = df.drop(['hotel'],axis=1)
y = df.loc[:,['hotel']]
```

In [23]:
```
x.shape
```

```
(119390, 31)
```

In [24]:
```
y.shape
```

```
(119390, 1)
```

Our Dataset has nominal as well as ordinal features so lets try Hash Encdoing

In [29]:
```
from sklearn.feature_extraction import FeatureHasher
```

In [30]:
```
## hash encoding only supports string column typs so converting them to string
x_train_hash=x.copy()
for c in x.columns:
    x_train_hash[c]=x[c].astype('str')
```

In [31]:
```
## Hashing the x_train_hash values
hashing=FeatureHasher(input_type='string') ## creating FeatureHasher object
train=hashing.transform(x_train_hash.values) ## stroing hashed encoded values in train
```

In [32]:
```
print('train data set has got {} rows and {} columns'.format(train.shape[0],train.shape[1]))
```

```
train data set has got 119390 rows and 1048576 columns
```

we see that hashencoded has created 1048576 additional Columns which again make it harder to debug back and see which features has the most contribution to target Prediction which makes it usecase only for compitions and not much suitable for realworld applications

In [ ]:

In [ ]:

In [ ]:

In [ ]:

Comparing OneHot,LabelEncode,HashEncode

Hash Encoder has a advantage of handling NaN values which other encoding methods doesn't so quickly comparing the three encoding techniques with each other based on logistic regression accuracy on same dataset

In [33]:
```
## Creating a Logistic regression fucntion for Saving Time
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score
def logistic(X,y):
    X_train,X_test,y_train,y_test=train_test_split(X,y,random_state=42,test_size=0.2)
    lr=LogisticRegression()
    lr.fit(X_train,y_train)
    y_pre=lr.predict(X_test)
    print('Accuracy : ',accuracy_score(y_test,y_pre))
```

We will be Using Hotel Booking Dataset

In [34]:
```
df=pd.read_csv('hotel_bookings.csv')
```

In [35]:
```
df.drop(['company','agent'],axis=1,inplace=True)
```

In [36]:
```
df.dropna(inplace=True)
```

In [37]:
```
df.get_dtype_counts()
```

```
object     12
int64      16
float64     2
dtype: int64
```

Since now our Data is cleaned we will apply encoding

In [38]:
```
x = df.drop(['hotel'],axis=1)
y = df.loc[:,['hotel']]
```

In [39]:
```
## one hot encoding and stroing dataframe in onehotencoded
onehotencoded = pd.get_dummies(x,drop_first=True)
```

In [40]:
```
from sklearn.preprocessing import LabelEncoder
```

In [41]:
```
## label encoding and stroing dataframe in labelencoded
labelencoded = x.apply(LabelEncoder().fit_transform)
```

In [42]:
```
## Hashing works only string features so converting every column string column
for i in x.columns:
    x[i] = x[i].astype(str)

hashing=FeatureHasher(input_type='string')
hashencoded=hashing.transform(x.values)
```

In [43]:
```
hashencoded.shape
```

```
(118898, 1048576)
```

we see that hashencoded has created 1048576 additional Columns which again make it harder to debug back and see which features has the most contribution to target Prediction which makes it usecase only for compitions and not much suitable for realworld applications

Comparing All three Encoding Methods results

In [44]:
```
print(hashencoded.shape)
logistic(hashencoded,y)
```

```
(118898, 1048576)



Accuracy :  0.8990328006728343
```

In [45]:
```
print(onehotencoded.shape)
logistic(onehotencoded,y)
```

```
(118898, 1171)



Accuracy :  0.838645920941968
```

In [46]:
```
print(labelencoded.shape)
logistic(labelencoded,y)
```

```
(118898, 29)



Accuracy :  0.786417157275021
```

We see that Encoding plays a vital role in Prediction Accuracy so it depends on the purpose of the project and dataset to which Encoding technique to use.

if you just wanted Best accuracy you would have used HashEncoding/FeatureHasher

if you wanted good accuracy as well as limited column use other encoding techniques

Thank You