Encoding categorical variables is a vital step in preparing data for machine learning tasks. When dealing with categorical data, characterized by non-numeric values such as text or categories, it becomes necessary to transform them into a numerical format for compatibility with machine learning algorithms. Categorical feature encoding is the process of converting the categorical features into numeric features. Various widely-used categorical encoding techniques are available, each presenting its unique set of advantages and drawbacks.

Two types of categorical features exist,

* **Ordinal Features**
* **Nominal Features**

**Ordinal Features:**

* Ordinal features are the features that have inherent ordering.
* Eg: Ratings such as Good, Bad.

**Nominal Features:**

* Nominal features are the features that don’t have any inherent ordering as opposed to Ordinal features.
* Eg: Names of persons, gender, yes, or no.

**Need for categorical feature encoding**

* Categorical features must be encoded before feeding it to the model because many Machine Learning algorithms don’t support categorical features as their input.
* Machine Learning algorithms and Deep Learning algorithms would support only **numerical variables/ quantitative variables**.

**Ordinal Encoding Techniques**

* Label Encoding or Ordinal Encoding

**Label Encoding:**

Label encoding is suitable for categorical features with only two distinct categories. In this technique, each category is assigned a unique integer label. Categories are assigned integer values starting from 0.

For example, if you have an ordinal categorical feature “Size” with values “Small,” “Medium,” and “Large,” label encoding would convert it as follows:

Size Size\_encoded  
0 Small 2  
1 Medium 1  
2 Large 0  
3 Medium 1  
4 Small 2

As you can see, each unique category in the ‘Size’ column has been replaced with a unique integer label, where ‘Small’ is encoded as 2, ‘Medium’ as 1, and ‘Large’ as 0.

**Python Implementation:**

from sklearn.preprocessing import LabelEncoder  
  
# Sample dataset with a categorical column  
data = {'Size': ['Small', 'Medium', 'Large', 'Medium', 'Small']}  
df = pd.DataFrame(data)  
  
# Initialize the LabelEncoder  
label\_encoder = LabelEncoder()  
  
# Fit and transform the 'Size' column  
df['Size\_encoded'] = label\_encoder.fit\_transform(df['Size'])

In this example, we have a data frame df with a categorical column 'Size,' which contains values like 'Small,' 'Medium,' and 'Large.' We use Scikit-learn's LabelEncoder to encode the 'Size' column into numerical values.

**Pros:** Works well for features with two categories

**Cons:**Machine learning algorithms may misinterpret the integer labels as having mathematical significance

**When to use:**Categorical features with two categories

**Ordinal Encoding:**

Ordinal encoding is similar to label encoding but allows you to explicitly define the mapping between categories and integer labels. This is especially useful when there is a clear and predefined ordinal relationship. You manually specify the order of categories and map them to integers accordingly.

Let’s look at it using an example in python

**Python Implementation:**

import pandas as pd  
import category\_encoders as ce  
  
# Sample dataset with an ordinal categorical column  
data = {'Education\_Level': ['High School', 'Bachelor\'s', 'Master\'s', 'Bachelor\'s', 'High School']}  
df = pd.DataFrame(data)  
  
# Define the order of categories  
education\_order = ['High School', 'Bachelor\'s', 'Master\'s']  
  
# Initialize the OrdinalEncoder with specified order  
ordinal\_encoder = ce.OrdinalEncoder(mapping=[{'col': 'Education\_Level', 'mapping': {level: index for index, level in enumerate(education\_order)}}])  
  
# Fit and transform the DataFrame  
df\_encoded = ordinal\_encoder.fit\_transform(df)  
  
# Display the DataFrame with ordinal encoding  
print(df\_encoded)

In this example, we use the OrdinalEncoder class from [category\_encoders](https://contrib.scikit-learn.org/category_encoders/index.html" \t "_blank)[library.](https://contrib.scikit-learn.org/category_encoders/index.html) This library provides a convenient way to apply various encoding techniques to your categorical data.

1. We start with a DataFrame df containing an ordinal categorical column 'Education\_Level' with values like 'High School,' 'Bachelor's,' and 'Master's.'
2. We define the order of categories in the education\_order list. The order should match the ordinal relationship of the categories.
3. We initialize the OrdinalEncoder with the specified mapping using the mapping parameter. We provide a dictionary specifying the column to encode ('Education\_Level') and a mapping dictionary that maps each category to its ordinal value based on the defined order.
4. We fit and transform the DataFrame using the ordinal\_encoder.

Education\_Level Education\_Level\_encoded  
0 High School 0  
1 Bachelor's 1  
2 Master's 2  
3 Bachelor's 1  
4 High School 0

**Pros:** Allows the user to explicitly specify the order in case of ordinal variables

**Cons:**Not applicable for non-ordinal variables

**When to use:**The best option for ordinal features

**Nominal Encoding Techniques**

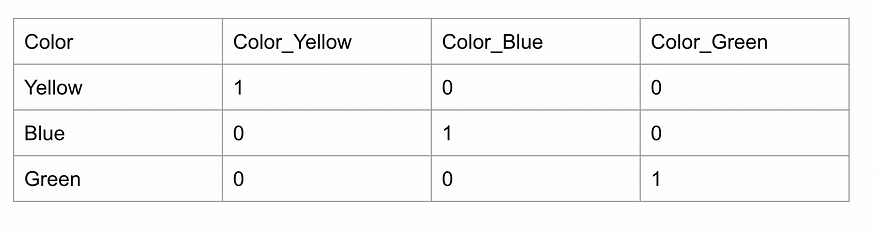
1. One-hot encoding
2. Frequency encoding
3. Target encoding
4. Leave-one-out encoding
5. Catboost encoding

**1. One-Hot Encoding:**

One-hot encoding is the most widely used categorical encoding technique. It is suitable for nominal categorical variables, where the categories have no inherent order or relationship. The idea behind one-hot encoding is to represent each category as a binary vector. Here’s how it works:

* For each category in a categorical column, a new binary column is created
* The binary column will have a value of 1 if the class is present, else it will be zero

For example, if you have a categorical feature “Color” with values “Yellow”, “Blue,” and “Green,” one-hot encoding would convert it into three binary columns:



**Python Implementation:**

import pandas as pd  
  
# Sample dataset with a categorical column  
data = {'Color': ['Red', 'Blue', 'Green', 'Red', 'Green']}  
df = pd.DataFrame(data)  
  
# Perform one-hot encoding using Pandas  
one\_hot\_encoded = pd.get\_dummies(df, columns=['Color'])

In this example, we start with a simple data frame df that contains a categorical column 'Color.' We then use the pd.get\_dummies function to perform one-hot encoding specifically on the 'Color' column. You can also customize one-hot encoding using additional parameters of the pd.get\_dummies function ([documentation link](https://pandas.pydata.org/docs/reference/api/pandas.get_dummies.html)), such as prefix adding a prefix to the new columns and prefix\_sep specifying a separator between the prefix and the original category name. These parameters can help you make the column names more meaningful in your dataset.

**Pros:** It preserves all information about the categories and doesn’t introduce any ordinal relationship

**Cons:**It can lead to a high dimensionality problem when dealing with a large number of classes

**When to use:**Ideally for categorical features with less than 10 categories (max 50 categories can be considered)

Things to note: Dummy encoding is very similar to one-hot encoding but it creates n-1 columns for a categorical feature with n categories. It doesn’t create a column for the first category to avoid the [dummy variable trap](https://www.statology.org/dummy-variable-trap/). The dummy variable trap is mostly applicable to linear regression models and we should use dummy encoding when using these models. Simply set the parameter drop\_first as True in pd.get\_dummies function to use dummy encoding.

Dummy encoding belongs to the family of [contrast coding](https://bookdown.org/pingapang9/linear_models_bookdown/contrasts.html#alternative-coding-schemes) for categorical features. There are a number of other encoding techniques in this system like Forward Difference Coding, Backward Difference Coding, Helmert Coding, Reverse Helmert Coding etc. For more details, you can refer [here](https://stats.oarc.ucla.edu/r/library/r-library-contrast-coding-systems-for-categorical-variables/#backward).

**4. Count Encoding:**

Count encoding or frequency encoding, replaces each category with the count of how many times it appears in the dataset. This encoding technique can be useful when there’s a correlation between the frequency of a category and the target variable.

Let’s look at it using an example in python

**Python Implementation:**

import pandas as pd  
from sklearn.model\_selection import train\_test\_split  
import category\_encoders as ce  
  
# Generate a dummy dataset with categorical variables  
data = {  
 'Color': ['Red', 'Blue', 'Green', 'Red', 'Red', 'Blue', 'Green'],  
 'Size': ['Small', 'Medium', 'Large', 'Medium', 'Small', 'Small', 'Medium'],  
 'Label': [1, 0, 1, 1, 0, 0, 1]  
}  
  
df = pd.DataFrame(data)  
  
# Split the data into training and test sets  
train\_df, test\_df = train\_test\_split(df, test\_size=0.2, random\_state=42)  
  
# Initialize the CountEncoder  
count\_encoder = ce.CountEncoder()  
  
# Fit the encoder on the training data  
count\_encoder.fit(train\_df[['Color', 'Size']])  
  
# Transform both the training and test datasets  
train\_encoded = count\_encoder.transform(train\_df[['Color', 'Size']])  
test\_encoded = count\_encoder.transform(test\_df[['Color', 'Size']])  
  
# Display the encoded datasets  
print("Training Data (After Count Encoding):\n", train\_encoded)  
print("\nTest Data (After Count Encoding):\n", test\_encoded)

In this example, we use the category\_encoders.CountEncoder() to initialize the encoder. The encoder is then fitted on the training data using count\_encoder.fit(train\_df[['Color', 'Size']]), and the transformation is applied to both the training and test datasets using count\_encoder.transform(train\_df[['Color', 'Size']]) and count\_encoder.transform(test\_df[['Color', 'Size']]), respectively. The resulting datasets (train\_encoded and test\_encoded) now contain count-encoded values for the categorical variables.  
Make sure to install the category\_encoders library using:

pip install category\_encoders

**Pros:** It reduces dimensionality compared to one-hot encoding. Count encoding retains the original information about the frequency of each category in the dataset.

**Cons:**While count encoding preserves frequency information, it discards any other meaningful information or relationships that may exist between categories. Count encoding can be sensitive to data imbalances.

**When to use:**This encoding technique can be useful when there’s a correlation between the frequency of a category and the target variable. Also applicable for categorical features with a lot of categories. Also, the count\_encoder should be fit only on the train dataset. The fitted object should be used to transform test and out of time (OOT) datasets.

**Target Encoding (Mean Encoding)**

Target encoding, also known as mean encoding, involves replacing each category with the mean (or some other statistic) of the target variable for that category. Here’s how target encoding works:

* Calculate the mean of the target variable for each category.
* Replace the category with its corresponding mean value.

**Python Implementation:**

import pandas as pd  
from sklearn.model\_selection import train\_test\_split  
import category\_encoders as ce  
  
# Generate a dummy dataset with categorical variables  
data = {  
 'Color': ['Red', 'Blue', 'Green', 'Red', 'Red', 'Blue', 'Green'],  
 'Size': ['Small', 'Medium', 'Large', 'Medium', 'Small', 'Small', 'Medium'],  
 'Label': [1, 0, 1, 1, 0, 0, 1]  
}  
  
df = pd.DataFrame(data)  
  
# Split the data into training and test sets  
train\_df, test\_df = train\_test\_split(df, test\_size=0.2, random\_state=42)  
  
# Initialize the MeanEncoder  
mean\_encoder = ce.TargetEncoder()  
  
# Fit the encoder on the training data  
mean\_encoder.fit(train\_df[['Color', 'Size']], train\_df['Label'])  
  
# Transform both the training and test datasets  
train\_encoded = mean\_encoder.transform(train\_df[['Color', 'Size']])  
test\_encoded = mean\_encoder.transform(test\_df[['Color', 'Size']])  
  
# Display the encoded datasets  
print("Training Data (After Mean Encoding):\n", train\_encoded)  
print("\nTest Data (After Mean Encoding):\n", test\_encoded)

In this example, we use the category\_encoders.TargetEncoder() to initialize the encoder. The encoder is then fitted on the training data using mean\_encoder.fit(train\_df[['Color', 'Size']], train\_df['Label']), and the transformation is applied to both the training and test datasets using mean\_encoder.transform(train\_df[['Color', 'Size']]) and mean\_encoder.transform(test\_df[['Color', 'Size']]), respectively. The resulting datasets (train\_encoded and test\_encoded) now contain mean-encoded values for the categorical variables.

**Pros:** Target encoding leverages the relationship between categorical variables and the target variable, making it a powerful encoding technique when this relationship is significant. It retains the information within the original feature, making it memory-efficient.

**Cons:**One of the significant drawbacks of target encoding is the potential for overfitting, especially when applied to small datasets. It suffers from the problem of target leakage as the target variable is used to directly encode the input feature and the same feature is used to fit a model on the target variable.

**When to use:**It is suitable for categorical features exhibiting a high number of categories. In the context of multi-class classification tasks, the initial step involves employing one-hot encoding on the target variable. This results in n binary columns, each corresponding to a specific class of the target variable. However, it’s noteworthy that only n-1 of these binary columns are linearly independent. As a consequence, any one of these columns can be omitted. Subsequently, the standard target encoding procedure is applied to each categorical feature, utilizing each binary label individually. Consequently, for a single categorical feature, n-1 target-encoded features are generated. If there are k categorical features in the dataset, the cumulative result is k times (n-1) features.

There are other variations of such target-based encoding methods like [WOE encoding, and M-Estimate Encoder](https://towardsdatascience.com/benchmarking-categorical-encoders-9c322bd77ee8) which have the same pros and cons.

**Leave-one-out Encoding**

Leave-One-Out Encoding (LOO Encoding) is a method that encodes each category in a categorical variable by calculating the mean of the target variable, excluding the current data point, which belongs to that category.

Here’s how LOO encoding works:

For each category within a categorical variable:

* Remove the current data point (row) from consideration.
* Calculate the mean of the target variable (usually binary, 0 or 1 for classification or a continuous value for regression) for the remaining data points in the same category. For multi-class classification, same steps as target encoding need to be followed.
* Assign this mean value as the encoding for the current data point’s category.

**Python Implementation**

import category\_encoders as ce  
import pandas as pd  
from sklearn.model\_selection import train\_test\_split  
  
# Sample dataset (replace this with your own dataset)  
data = {  
 'Category1': ['A', 'B', 'A', 'B', 'A', 'A', 'B', 'B'],  
 'Category2': ['X', 'Y', 'X', 'Y', 'X', 'Y', 'X', 'Y'],  
 'Target': [1, 0, 1, 0, 1, 1, 0, 0]  
}  
  
df = pd.DataFrame(data)  
  
# Split the data into train and test sets  
train\_df, test\_df = train\_test\_split(df, test\_size=0.2, random\_state=42)  
  
# Specify categorical columns for LOO encoding  
categorical\_columns = ['Category1', 'Category2']  
  
# Initialize LOO encoder  
loo\_encoder = ce.leave\_one\_out.LeaveOneOutEncoder(cols=categorical\_columns)  
  
# Fit and transform on the training data  
train\_encoded = loo\_encoder.fit\_transform(train\_df, train\_df['Target'])  
  
# Transform the test data using the encoder fitted on the training data  
test\_encoded = loo\_encoder.transform(test\_df)  
  
# Display the results  
print("Encoded Training Data:")  
print(train\_encoded)  
  
print("\nEncoded Test Data:")  
print(test\_encoded)

Make sure to replace the sample dataset with your actual dataset. In this example, categorical\_columns is a list containing the names of the categorical columns that need encoding. The LeaveOneOutEncoder is then used with these specified columns. The encoder is fitted to the training data and used to transform both the training and test data.

**Pros:**LOO encoding can help mitigate data leakage by excluding the current data point from the calculation. It ensures that the encoding is less affected by that data point. LOO encoding retains information about the relationship between each category and the target variable.

**Cons:**LOO encoding can introduce variability into the encoded values, especially when dealing with categories with a small number of data points. The mean value can fluctuate significantly based on the exclusion of a single data point. In cases with small sample sizes, LOO encoding may result in overfitting, as it calculates the mean for each category based on limited data points. It is also computationally intensive.

**When to Use:** Use for categorical features with a large number of categories.

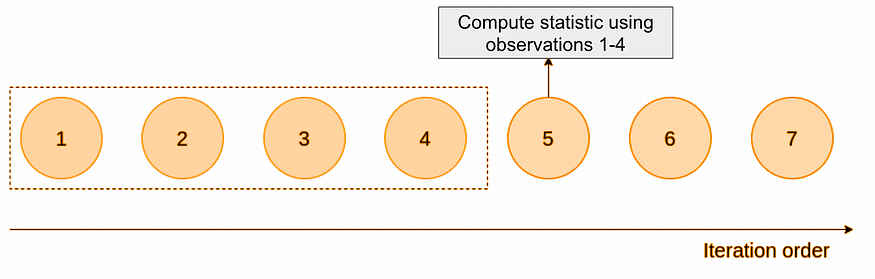
There is another variation of such a type of encoding called K-fold encoding. The only difference between LOO and K-fold is that instead of the single record being dropped, the dataset is split into equal-sized parts, and the entire part (to which the record belongs) is dropped when calculating the target mean.

**Catboost Encoder**

CatBoost is a popular tree-based model developed by Yandex researchers and it handles categorical variables out of the box (hence the name of the algorithm). It is one of the best options we have to encode categorical features with high cardinality. Catboost encoder is based on a concept called **Ordered Target Statistics.**

The method of transforming categorical features into numerical ones includes the following stages:

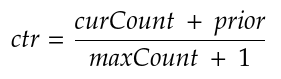
1. We draw a random permutation order of the dataset
2. For regression problems, the target variable is transformed from floating numbers to a set of integers using [quantization](https://catboost.ai/en/docs/concepts/quantization)
3. Then we iterate sequentially throughout the observations respecting that new order. For every observation, we compute a statistic of interest **(to be discussed in the next section)** using only the observations that we have already seen in the past. The random permutation acts as an artificial time.



4. The initial observations lack sufficient training data to generate reliable estimates, resulting in significant variability. To address this challenge, the creators of CatBoost suggested generating multiple random permutations and creating an encoding for each permutation. The ultimate outcome is obtained by averaging these distinct encodings.

**How the statistic of interest is calculated?**

The statistic calculation depends on the [selected strategy](https://catboost.ai/en/docs/concepts/algorithm-main-stages_cat-to-numberic) but the idea remains approximately the same. For example, with the strategy “BinarizedTargetMeanValue”, the following computation takes place:



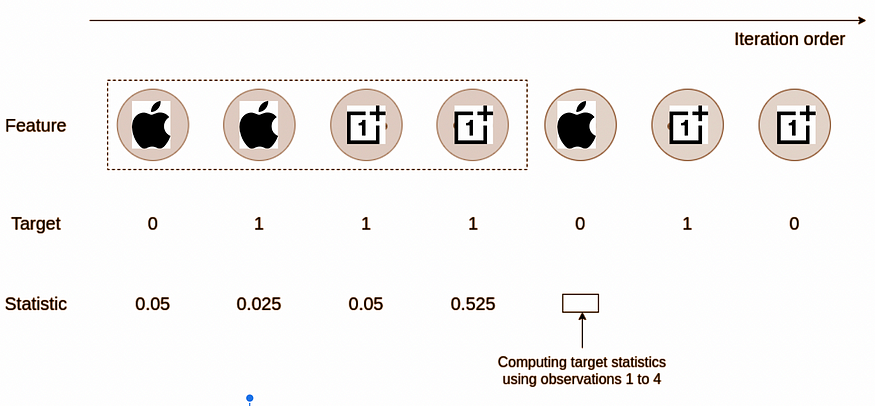
To understand the constituents of this formula, the best thing is to quote the documentation:

* ***curCount***is how many times the label value was equal to 1 with the current categorical feature value.
* ***maxCount***is the total number of objects (up to the current one) that have a categorical feature value matching the current one.
* ***prior****is a number (constant) defined by the starting parameters.*

Let’s take an example to understand the statistic calculation:

1. We have a dataset where we have a categorical feature of mobile brands. There are 2 brands: Apple and Oneplus.

2. We set the prior to 0.05.



When computing the statistic for the fifth observation, we have:

* countInClass = 1 (The number of times, in the past training data, the target was equal to 1 when the categorical feature was equal to the Apple)
* maxCount = 2 (The total number of Apple brands in the past training data)

The statistic for that fifth observation is then equal to (1+0.05)/(2+1)=0.35

**Python Implementation:**

import pandas as pd  
from category\_encoders.cat\_boost import CatBoostEncoder  
  
# Sample training and testing datasets with a categorical column  
train\_data = pd.DataFrame({'Category': ['A', 'B', 'C', 'A', 'B'], 'Target': [1, 0, 1, 0, 1]})  
test\_data = pd.DataFrame({'Category': ['A', 'C', 'B']})  
  
# Initialize the CatBoostEncoder  
catboost\_encoder = CatBoostEncoder()  
  
# Fit and transform the training data  
train\_encoded = catboost\_encoder.fit\_transform(train\_data['Category'], train\_data['Target'])  
  
# Transform the testing data  
test\_encoded = catboost\_encoder.transform(test\_data['Category'])

Please note that the behavior of the transformer would differ in transform and fit\_transform methods depending on whether the target variable is passed. If no target is passed, then the encoder will map the last value of the running mean to each category. If the target variable is passed then it will map all values of the running mean to each category’s occurrences.

Pros: It is the best method for encoding high cardinality categorical features as it minimizes target leakage and has no issue of increased dimensionality

Cons: For test and OOT data, it simply maps the last value of the running mean to each category.

When to use: Best choice for high cardinality categorical features

For a more detailed explanation, please refer [here](https://towardsdatascience.com/how-catboost-encodes-categorical-variables-3866fb2ae640).

There is another method that is effective in handling both dimensionality curse and target leakage called Bayesian target encoding. You can learn more from the tutorial below

The category encoder package still doesn’t have an option for Bayesian target encoding.

**How to handle new categories in the future?**

Sometimes in practical business problems, we end up having new categories being added to a categorical feature in the future. For example, if we have a feature called mobile brand and currently it has only 4 values (Apple, Oneplus, Mi, Vivo). But later on, a new brand was added to the data (Oppo). In this scenario, our encoder will throw an error as it doesn’t know how to handle the new mobile brand. It is also impractical to retrain the model every time a new category comes in. What do we do?

In situations like these where there is a possibility of a new category to be added in the future, it is best practice to not use the original feature and engineer a new feature with a category called “others”. How the feature is generated is based on business context and/or frequency distribution of categories. Let’s look at both.

Suppose I am building a fraud model, then fraudsters usually don’t use expensive brands like Apple or Oneplus, so I can create a new “high\_value\_mobile\_brand’ feature that has the value “Apple” and “Oneplus” if those brands are present else it has “others”. In this case any new mobile brand will be tagged as “others” and the encoder will work. If there is no business angle based on which the feature can be generated, we can tag the category with very few counts as “others”.

Another angle to keep in mind is if we use one-hot encoding/dummy encoding, we need to ensure the future datasets have all the categories used in the train data. If they are not present the model will throw an error. Then we need to add additional columns for the categories that are absent.

**Conclusion**

In conclusion, encoding categorical variables is an essential preprocessing step in preparing data for machine learning applications. Categorical data, often represented by non-numeric values like text or categories, requires transformation into a numerical format to be compatible with machine learning algorithms. This blog has explored various widely used categorical encoding techniques, each with its own strengths and limitations. Understanding the nuances of these encoding techniques is crucial for making informed decisions in data preprocessing. There are a few other encoding methods like the James-Stein estimator. It has, however, one practical limitation — it was defined only for normal distributions. Hence, we skipped it.

It’s also important to note that proper handling of new categories in the future is essential, especially in practical business scenarios. Strategies such as creating an “others” category or engineering features based on business context can help address the challenge of new category additions without requiring model retraining.

In summary, the effective encoding of categorical variables contributes significantly to the success of machine learning models by ensuring that valuable information within these variables is appropriately represented for predictive modeling.

[All you need to know about encoding techniques! | by Indraneel Dutta Baruah | ANOLYTICS | Medium](https://medium.com/anolytics/all-you-need-to-know-about-encoding-techniques-b3a0af68338b)

[Different types of Encoding Techniques in Machine Learning | by Abdullah Zunorain | Medium](https://abdullahzunorain.medium.com/encoding-techniques-in-machine-learning-0d40bc76bb48)

[Categorical Data Encoding Techniques | by Krishnakanth Naik Jarapala | AI Skunks | Medium](https://medium.com/aiskunks/categorical-data-encoding-techniques-d6296697a40f)

[Feature Encoding Techniques - Machine Learning - GeeksforGeeks](https://www.geeksforgeeks.org/feature-encoding-techniques-machine-learning/)

[Categorical Feature Encoding Techniques (kaggle.com)](https://www.kaggle.com/code/srivignesh/categorical-feature-encoding-techniques)