Benchmarking Categorical Encoders

Denis Vorotyntsev Jul 14, 2019 · 16 min read ★

Most tabular datasets contain categorical features. The simplest way to work with these is to encode them with Label Encoder. It is simple, yet sometimes not accurate.

different strategies to encode categorical variables. Then I will show you how those could be improved through Single and Double Validation. The final part of the paper is devoted to the discussion of benchmarks results (which also could be found in my GitHub repo — CategoricalEncodingBenchmark).

The following material describes binary classification task, yet all formulas and approaches could be applied to multiclass classification (as far as it could be represented as a binary classification) and regression.

TL; DR

1. There is no free lunch. You have to try multiple types of encoders to find the best one for your data;
2. However, the most stable and accurate encoders are target-based encoders with Double Validation: Catboost Encoder, James-Stein Encoder, and Target Encoder;
3. *encoder.fit_transform()* on the whole train is a road to nowhere: it turned out that Single Validation is a much better option than commonly used None Validation. If you want to achieve a stable high score, Double Validation is your choice, but bear in mind that it requires much more time to be trained;
4. Regularization is a must for target-based encoders.

Categorical Encoders

If you are looking for a better understanding of categorical encoding, I recommend you to grab a pen and some paper and make your own calculations with the formulas I provided below. It wouldn't take much time, but it is really helpful. In the formulas, I'm going to use the following parameters:

- y and $y+$ — the total number of observations and the total number of positive observations ($y=1$);
- xi, yi — the *i-th* value of category and target;
- n and $n+$ — the number of observations and the number of positive observations ($y=1$) for a given value of a categorical column;

The example train dataset looks like this:

	category	target
0	A	0
1	A	1
2	A	0
3	A	1
4	B	0
5	B	1
6	B	0
7	C	1
8	C	0
9	D	1

	category	target
0	A	None
1	B	None
2	C	None
3	D	None
4	NewCategory	None

Example train dataset on the left and test dataset on the right

- $y=10, y+=5;$
- $ni=$"D", $yi=1$ for 9th line of the dataset (the last observation);
- For category B: $n=3, n+=1;$
- $prior = y+/y = 5/10 = 0.5$.

With this in mind, let's start from simple ones while gradually increasing the encoder's complexity.

Label Encoder (LE) or Ordinal Encoder (OE)

	category	category_representation	target
0	A	1	0
1	A	1	1
2	A	1	0
3	A	1	1
4	B	2	0
5	B	2	1
6	B	2	0
7	C	3	1

	category	category_representation	target
0	A	1.0	None
1	B	2.0	None
2	C	3.0	None
3	D	4.0	None
4	NewCategory	-1.0	None

Category representation — Label Encoding

The most common way to deal with categories is to simply map each category with a number. By applying such transformation, a model would treat categories as ordered integers, which in most cases is wrong. Such transformation should not be used “as is” for several types of models (Linear Models, KNN, Neural Nets, etc.). While applying gradient boosting it could be used only if the type of a column is specified as *“category”*:

```
df[“category_representation”] =
df[“category_representation”].astype(“category”)
```

New categories in Label Encoder are replaced with “-1” or None. If you are working with tabular data and your model is gradient boosting (especially LightGBM library), LE is the simplest and efficient way for you to work with categories in terms of memory (the category type in python consumes much less memory than the object type).

One-Hot-Encoder (OHE) (dummy encoding)

	category	category_representation_1	category_representation_2	category_representation_3	category_representation_4	target
0	A	1	0	0	0	0
1	A	1	0	0	0	1
2	A	1	0	0	0	0
3	A	1	0	0	0	1
4	B	0	1	0	0	0
5	B	0	1	0	0	1
6	B	0	1	0	0	0
7	C	0	0	1	0	1
8	C	0	0	1	0	0
9	D	0	0	0	1	1

	category	category_representation_1	category_representation_2	category_representation_3	category_representation_4	target
0	A	1	0	0	0	None
1	B	0	1	0	0	None
2	C	0	0	1	0	None
3	D	0	0	0	1	None
4	NewCategory	0	0	0	0	None

Category representation — One-Hot-Encoding

multiple columns. The numbers are replaced by 1s and 0s depending on which column has what value.

OHE expands the size of your dataset, which makes it memory-inefficient encoder. There are several strategies to overcome the memory problem with OHE, one of which is working with sparse not dense data representation.

Sum Encoder (Deviation Encoding or Effect Encoding)

	category	category_representation_0	category_representation_1	category_representation_2	target
0	A	1.0	0.0	0.0	0
1	A	1.0	0.0	0.0	1
2	A	1.0	0.0	0.0	0
3	A	1.0	0.0	0.0	1
4	B	0.0	1.0	0.0	0
5	B	0.0	1.0	0.0	1
6	B	0.0	1.0	0.0	0
7	C	0.0	0.0	1.0	1
8	C	0.0	0.0	1.0	0
9	D	-1.0	-1.0	-1.0	1

	category	category_representation_0	category_representation_1	category_representation_2	target
0	A	1.0	0.0	0.0	None
1	B	0.0	1.0	0.0	None
2	C	0.0	0.0	1.0	None
3	D	-1.0	-1.0	-1.0	None
4	NewCategory	-1.0	-1.0	-1.0	None

Category representation — Sum Encoding

Sum Encoder compares the mean of the dependent variable (target) for a given level of a categorical column to the overall mean of the target. Sum Encoding is very similar to OHE and both of them are commonly used in Linear Regression (LR) types of models.

However, the difference between them is the interpretation of LR coefficients: whereas in OHE model the intercept represents the mean for the baseline condition and coefficients represents simple effects (the difference between one particular condition and the baseline), in Sum Encoder model the intercept represents the grand mean

Helmert Encoder

	category	category_representation_0	category_representation_1	category_representation_2	category_representation_3	category_representation_4	target
0	A	-1.0	-1.0	-1.0	-1.0	-1.0	0
1	A	-1.0	-1.0	-1.0	-1.0	-1.0	1
2	A	-1.0	-1.0	-1.0	-1.0	-1.0	0
3	A	-1.0	-1.0	-1.0	-1.0	-1.0	1
4	B	1.0	-1.0	-1.0	-1.0	-1.0	0
5	B	1.0	-1.0	-1.0	-1.0	-1.0	1
6	B	1.0	-1.0	-1.0	-1.0	-1.0	0
7	C	0.0	2.0	-1.0	-1.0	-1.0	1
8	C	0.0	2.0	-1.0	-1.0	-1.0	0
9	D	0.0	0.0	3.0	-1.0	-1.0	1

	category	category_representation_0	category_representation_1	category_representation_2	category_representation_3	category_representation_4	target
0	A	-1.0	-1.0	-1.0	-1.0	-1.0	None
1	B	1.0	-1.0	-1.0	-1.0	-1.0	None
2	C	0.0	2.0	-1.0	-1.0	-1.0	None
3	D	0.0	0.0	3.0	-1.0	-1.0	None
4	NewCategory	0.0	0.0	3.0	-1.0	-1.0	None

Category representation — Helmert Encoder

Helmert coding is a third commonly used type of categorical encoding for regression along with OHE and Sum Encoding. It compares each level of a categorical variable to the mean of the subsequent levels. Hence, the first contrast compares the mean of the dependent variable for “A” with the mean of all of the subsequent levels of categorical column (“B”, “C”, “D”), the second contrast compares the mean of the dependent variable for “B” with the mean of all of the subsequent levels (“C”, “D”), and the third contrast compares the mean of the dependent variable for “C” with the mean of all of the subsequent levels (in our case only one level — “D”).

This type of encoding can be useful in certain situations where levels of the categorical variable are ordered, say, from lowest to highest, or from smallest to largest.

Frequency Encoder

1	A	4	1	1	B	3	None
2	A	4	0	2	C	2	None
3	A	4	1	3	D	1	None
4	B	3	0	4	NewCategory	1	None
5	B	3	1				
6	B	3	0				
7	C	2	1				
8	C	2	0				
9	D	1	1				

Category representation — Frequency Encoding

Frequency Encoding counts the number of a category's occurrences in the dataset. New categories in test dataset encoded with either "1" or counts of category in a test dataset, which makes this encoder a little bit tricky: encoding for different sizes of test batch might be different. You should think about it beforehand and make preprocessing of the train as close to the test as possible.

To avoid such problem, you might also consider using a Frequency Encoder variation — Rolling Frequency Encoder (RFE). RFE counts the number a category's occurrences for the last *dt* timesteps from a given observation (for example, for *dt*= 24 hours).

Nevertheless, Frequency Encoding and RFE are especially efficient when your categorical column has "long tails", i.e. several frequent values and the remaining ones have only a few examples in the dataset. In such a case, Frequency Encoding would catch the similarity between rare columns.

Target Encoder (TE)

	category	category_representation	target		category	category_representation	target
0	A	0.5000	0	0	A	0.5000	None
1	A	0.5000	1	1	B	0.3532	None
2	A	0.5000	0	2	C	0.5000	None
3	A	0.5000	1	3	D	0.5000	None
4	B	0.3532	0	4	NewCategory	0.5000	None
5	B	0.3532	1				
6	B	0.3532	0				
7	C	0.5000	1				

Category representation — Target Encoding

Target Encoding has probably become the most popular encoding type because of Kaggle competitions. It takes information about the target to encode categories, which makes it extremely powerful. The encoded category values are calculated according to the following formulas:

$$s = \frac{1}{1 + exp(-\frac{n-mdl}{a})}$$

$$\hat{x}^k = prior * (1 - s) + s * \frac{n^+}{n}$$

Here, *mdl* — min data (samples) in leaf, *a* — smoothing parameter, representing the power of regularization. Recommended values for *mdl* and *a* are in the range of 1 to 100. New values of category and values with just single appearance in train dataset are replaced with the prior ones.

Target Encoder is a powerful tool, yet it has a huge disadvantage — target leakage: it uses information about the target. Because of the target leakage, model overfits the training data which results in unreliable validation and lower test scores. To reduce the effect of target leakage, we may increase regularization (it's hard to tune those hyperparameters without unreliable validation), add random noise to the representation of the category in train dataset (some sort of augmentation), or use Double Validation.

M-Estimate Encoder

	category	category_representation	target
0	A	0.416667	0
1	A	0.416667	1
2	A	0.416667	0
3	A	0.416667	1
4	B	0.250000	0
5	B	0.250000	1

	category	category_representation	target
0	A	0.416667	None
1	B	0.250000	None
2	C	0.250000	None
3	D	0.250000	None
4	NewCategory	0.250000	None

8	C	0.250000	0
9	D	0.250000	1

Category representation — M-Estimate Encoder

M-Estimate Encoder is a simplified version of Target Encoder. It has only one hyperparameter — m, which represents the power of regularization. The higher value of m results into stronger shrinking. Recommended values for m is in the range of 1 to 100.

$$\hat{x}^k = \frac{n^+ + prior * m}{y^+ + m}$$

In different sources, you may find another formula of M-Estimator. Instead of $y+$ there is n in the denominator. I found that such representation has similar scores.

UPD (17.07.2019): The formula for M-Estimate Encoder in Categorical Encoders library contained a bug. The right one should have n in the denominator. However, both approaches show pretty good scores. The following benchmark is done via "wrong" formula.

Weight Of Evidence Encoder (WOE)

	category	category_representation	target
0	A	0.000000	0
1	A	0.000000	1
2	A	0.000000	0
3	A	0.000000	1
4	B	-0.405465	0
5	B	-0.405465	1
6	B	-0.405465	0
7	C	0.000000	1
8	C	0.000000	0
9	D	0.000000	1

	category	category_representation	target
0	A	0.000000	None
1	B	-0.405465	None
2	C	0.000000	None
3	D	0.000000	None
4	NewCategory	0.000000	None

Category representation — Weight Of Evidence Encoder

calculated from the basic odds ratio:

```
a = Distribution of Good Credit Outcomes
b = Distribution of Bad Credit Outcomes
WoE = ln(a / b)
```

However, if we use formulas as is, it might lead to target leakage and overfit. To avoid that, regularization parameter a is induced and WoE is calculated in the following way:

$$nominator = \frac{n^+ + a}{y^+ + 2 * a}$$

$$denominator = \frac{n - n^+ + a}{y - y^+ + 2 * a}$$

$$\hat{x}^k = ln(\frac{nominator}{denominator})$$

James-Stein Encoder

	category	category_representation	target
0	A	0.500000	0
1	A	0.500000	1
2	A	0.500000	0
3	A	0.500000	1
4	B	0.363636	0
5	B	0.363636	1
6	B	0.363636	0
7	C	0.500000	1
8	C	0.500000	0
9	D	1.000000	1

	category	category_representation	target
0	A	0.500000	None
1	B	0.363636	None
2	C	0.500000	None
3	D	1.000000	None
4	NewCategory	1.000000	None

Category representation — James-Stein Encoder Encoder

simplified Stein's original Gaussian random vectors mean estimation method of 1956. Stein and James proved that a better estimator than the "perfect" (i.e. mean) estimator exists, which seems to be somewhat of a paradox. However, the James-Stein estimator outperforms the sample mean when there are *several* unknown populations means — not just one.

The idea behind James-Stein Encoder is simple. Estimation of the mean target for category *k* could be calculated according to the following formula:

$$\hat{x}^k = (1 - B) * \frac{n^+}{n} + B * \frac{y^+}{y}$$

Encoding is aimed to improve the estimation of the category's mean target (first member of the amount) by shrinking them towards a more central average (second member of the amount). The only hyperparameter in the formula is *B* — the power of shrinking. It could be understood as the power of regularization, i.e. the bigger values of *B* will result in the bigger weight of global mean (underfit), while the lower values of *B* are, the bigger weight of condition mean (overfit).

One way to select *B* is to tune it like a hyperparameter via cross-validation, but Charles Stein came up with another solution to the problem:

$$B = \frac{\mathrm{Var}[y^k]}{\mathrm{Var}[y^k] + \mathrm{Var}[y]}$$

Intuitively, the formula can be seen in the following sense: if we could not rely on the estimation of category mean to target (it has high variance), it means we should assign a bigger weight to the global mean.

Wait, but how we could trust the estimation of variance if we could not rely on the estimation of the mean? Well, we may either say that the variance among all categories is the same and equal to the global variance of *y* (which might be a good estimation, if we don't have too many unique categorical values; it is called *pooled variance* or *pooled model*) or replace the variances with squared standard errors, which penalize small observation counts (*independent model*).

we can either convert binary targets with a log-odds ratio as it was done in WoE Encoder (which is used by default because it is simple) or use beta distribution.

Leave-one-out Encoder (LOO)

	category	category_representation	target
0	A	0.666667	0
1	A	0.333333	1
2	A	0.666667	0
3	A	0.333333	1
4	B	0.500000	0
5	B	0.000000	1
6	B	0.500000	0
7	C	0.000000	1
8	C	1.000000	0
9	D	0.500000	1

	category	category_representation	target
0	A	0.500000	None
1	B	0.333333	None
2	C	0.500000	None
3	D	0.500000	None
4	NewCategory	0.500000	None

Category representation — Leave-one-out Encoding

Leave-one-out Encoding (LOO or LOOE) is another example of target-based encoders. The name of the method clearly speaks for itself: we calculate mean target of category *k* for observation *j* if observation *j* is removed from the dataset:

$$\hat{x}_i^k = \frac{\sum_{j \neq i}(y_j * (x_j == k)) - y_i}{\sum_{j \neq i} x_j == k}$$

While encoding the test dataset, a category is replaced with the mean target of the category *k* in the train dataset:

$$\hat{x}^k = \frac{\sum(y_j * (x_j == k))}{\sum x_j == k}$$

One of the problems with LOO, just like with all other target-based encoders, is target leakage. But when it comes to LOO, this problem gets really dramatic, as far as we may

$$t^k = \frac{\sum(y_j * (x_j == k)) - 0.5}{\sum x_j == k}$$

Another problem with LOO is a shift between values in the train and the test samples. You could observe it from the picture above. Possible values for category “A” in the train sample are 0.67 and 0.33, while in the test one — 0.5. It is a result of the different number of counts in train and test datasets: for category “A” denominator is equal to *n* for test and *n-1* for train dataset. Such a shift may gradually reduce the performance of tree-based models.

Catboost Encoder

	category	category_representation	target
0	A	0.500	0
1	A	0.250	1
2	A	0.500	0
3	A	0.375	1
4	B	0.500	0
5	B	0.250	1
6	B	0.500	0
7	C	0.500	1
8	C	0.750	0
9	D	0.500	1

	category	category_representation	target
0	A	0.500	None
1	B	0.375	None
2	C	0.500	None
3	D	0.500	None
4	NewCategory	0.500	None

Category representation — CatBoost Encoder

Catboost is a recently created target-based categorical encoder. It is intended to overcome target leakage problems inherent in LOO. In order to do that, the authors of Catboost introduced the idea of “time”: the order of observations in the dataset. Clearly, the values of the target statistic for each example rely only on the observed history. To calculate the statistic for observation *j* in train dataset, we may use only observations, which are collected before observation *j*, i.e. $i \leq j$:

$$\hat{x}_i^k = \frac{\sum_{j=0}^{j \leqslant i}(y_j * (x_j == k)) - y_i + prior}{\sum^{j \leqslant i} x_j == k}$$

To prevent overfitting, the process of target encoding for train dataset is repeated several times on shuffled versions of the dataset and results are averaged. Encoded values of the test data are calculated the same way as in LOO Encoder:

$$\hat{x}^k = \frac{\sum (y_j * (x_j == k)) + prior}{\sum x_j == k}$$

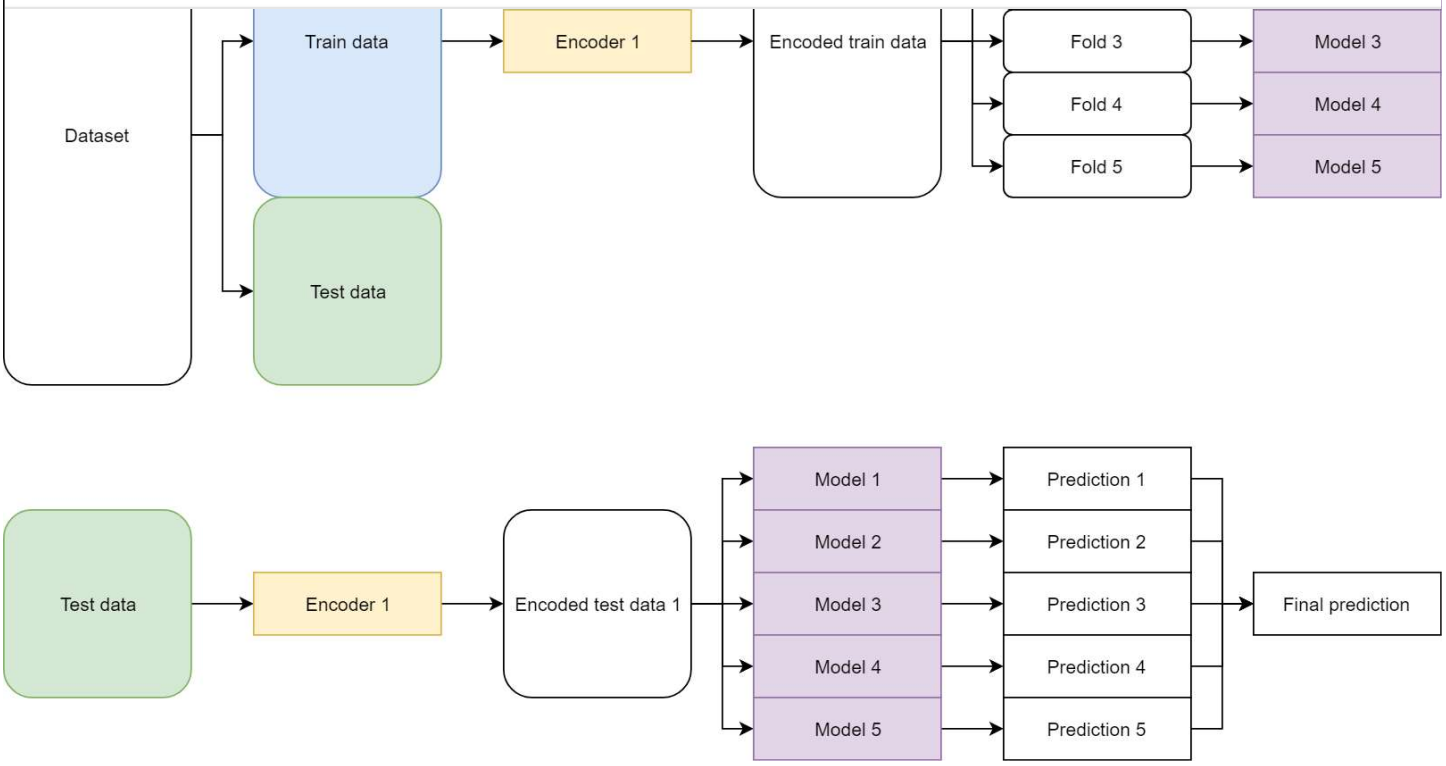
Catboost "on the fly" Encoding is one of the core advantages of CatBoost — library for gradient boosting, which showed state of the art results on several tabular datasets when it was presented by Yandex.

Validation Approaches

Model validation is probably the most important aspect of Machine Learning. While working with data that contains categorical variables, we may want to use one of the three types of validation. None Validation is the simplest one, yet least accurate. Double Validation could show great scores, but it is as slow as a turtle. And Single Validation is kind of a cross between the first two methods.

This section is devoted to the discussion of each validation type in details. For better understanding, for each type of validation, I added block diagrams of the pipeline.

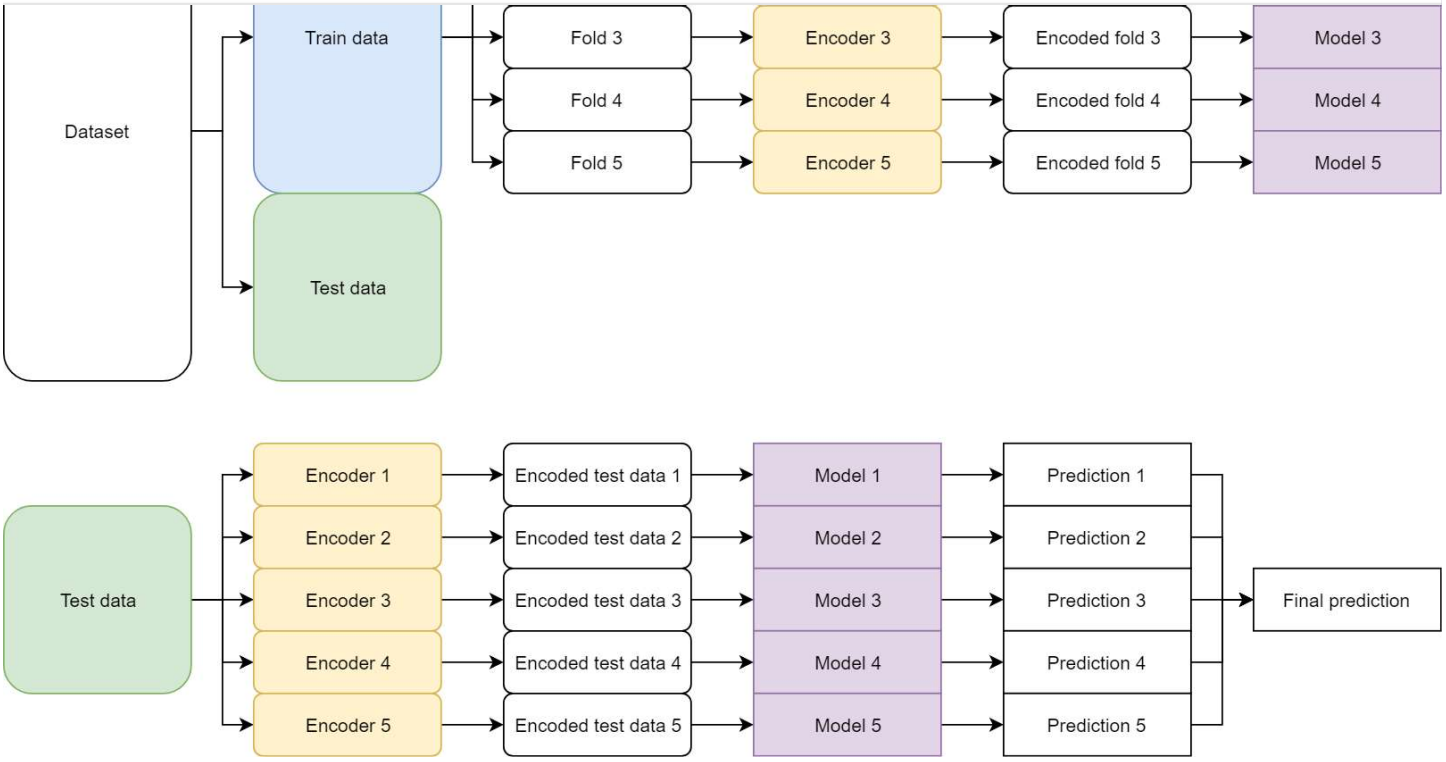
None Validation



In the case of None Validation, the single encoder is fitted to the whole train data. Validation and test parts of the dataset are processed with the same single encoder. Being the simplest one, this approach is wildly used, but it leads to overfitting during training and unreliable validation scores. The most obvious reasons for that are: for target encoders, None validation induces a shift between train and test part (see an example in LOO encoder); test dataset may contain new categorical values, but neither train nor validation sample contains them during training, that is why:

- The model doesn't know how to handle new categories;
- The optimal number of trees for train and test samples may be different.

Single Validation



Single Validation is a better way to carry out validation. By using Single Validation we overcome one of the problems of None Validation: the shift between validation and test datasets. Within this approach, we fit separate categorical encoders to each fold of training data. Validation part of each fold is processed the same way as the test data, so it became more similar to it, which positively affects hyperparameters tuning. The optimal hyperparameters obtained from Single Validation would be closer to optimal hyperparameters of test dataset then the ones obtained from None Validation.

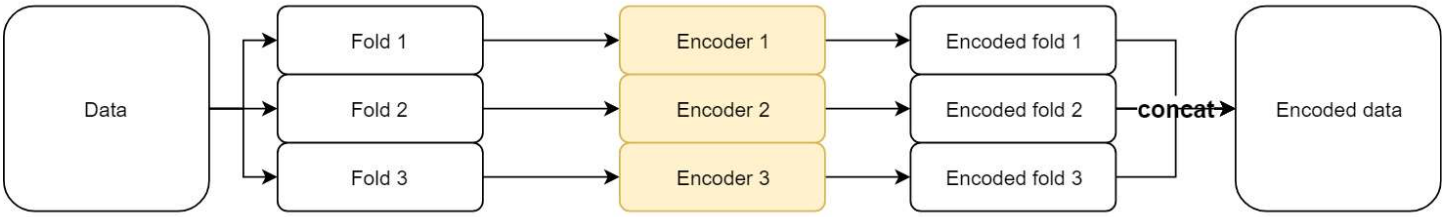
Note: while we are tuning hyperparameters, i.e. a number of trees, on validation part of the data the validation scores are going to be slightly overfitted anyway. So will be the test scores, because the type of encoder is also a hyperparameter of the pipeline. I reduced the overfitting effect by making test bigger (40% of the usual data).

The other good advantage of Single Validation is diversity across folds. By fitting encoder on a subset of the data, we achieve different representation of category, which positively affects the final blending of predictions.

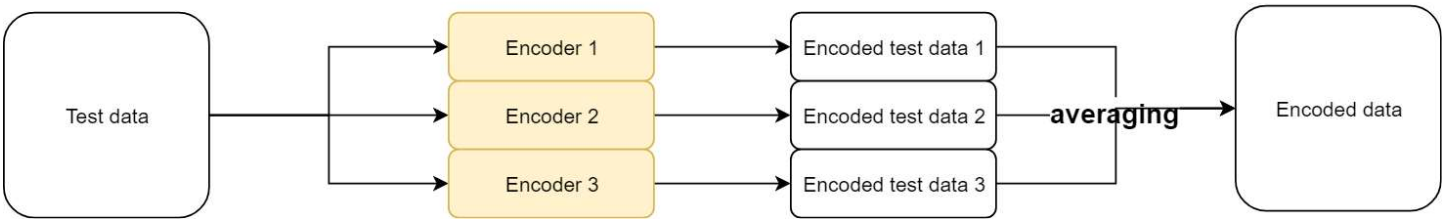
Double Validation

Double validation encoding:

During training:



During testing:



Double Validation is an extension of Single Validation. The pipeline for Double Validation is exactly the same as for Single Validation, except Encoder part: instead of the single encoder for each fold, we will use several encoders fitted on different subsets.

Double Validation aims to reduce the effect of the second problem of None Validation — new categories. It is achieved by splitting each fold into sub-folds, fitting separate encoder to each sub-fold, and then concating (in case of train part) or averaging (in case of validation or test parts) the results. By applying Double Validation we induce a noise (new categories) into training data, which works as augmentation of the data. However, Double Validation has two major disadvantages:

- We can not use it “as is” for encoders, which represents single category as multiple columns (for example, OHE, Sum Encoder, Helmert Encoder, and Backward Difference Encoder);
- The time complexity of Double Validation is approximately k times bigger than in Single validation (k — number of subfolds).

title of the paper. So, let's move from that to practice! In the following section, I'm going to show you how I determined the best categorical encoder for tabular data.

Datasets

I created and tested pipeline for categories benchmarking on the datasets which are presented in the table below. All datasets except *poverty_A(B, C)* came from different domains; they have a different number of observations, as well as of categorical and numerical features. The objective for all datasets is to perform binary classification. Preprocessing was quite simple: I removed all time-based columns from the datasets. The ones left were either categorical or numerical.

Table 1.1 Used datasets

Name	Total points	Train points	Test points	Number of features	Number of categorical features	Short description
Telecom	7.0k	4.2k	2.8k	20	16	Churn prediction for telecom data
Adult	48.8k	29.3k	19.5k	15	8	Predict if persons' income is bigger 50k
Employee	32.7k	19.6k	13.1k	10	9	Predict an employee's access needs, given his/her job role
Credit	307.5k	184.5k	123k	121	18	Loan repayment
Mortgages	45.6k	27.4k	18.2k	20	9	Predict if house mortgage is founded
Promotion	54.8	32.8k	21.9k	13	5	Predict if an employee will get a promotion
Kick	72.9k	43.7k	29.1k	32	19	Predict if a car purchased at auction is good/bad buy
Kdd_upselling	50k	30k	20k	230	40	Predict up-selling for a customer
Taxi	892.5k	535.5k	357k	8	5	Predict the probability of an offer being accepted by a certain driver
Poverty_A	37.6k	22.5k	15.0k	41	38	Predict whether or not a given household for a given country is poor or not
Poverty_B	20.2k	12.1k	8.1k	224	191	Predict whether or not a given household for a given country is poor or not
Poverty_C	29.9k	17.9k	11.9k	41	35	Predict whether or not a given household for a given country is poor or not

Pipeline

and it is used only once — for final scoring (scoring metric used — *ROC AUC*). No encoding on whole data, no pseudo labeling, no TTA, etc. were applied during the training or predicting stage. I wanted the experiments to be as close to possible production settings as possible.

After the train-test split, the training data was split into 5 folds with shuffle and stratification. After that, 4 of them were used for the fitting encoder (for a case of None Validation — encoder if fitted to the whole train dataset before splitting) and LightGBM model (LlightGBM — library for gradient boosting from Microsoft), and 1 more fold was used for early stopping. The process was repeated 5 times and in the end, we had 5 trained encoders and 5 LGB models. The LightGBM model parameters were as follows:

```
"metrics": "AUC",
"n_estimators": 5000,
"learning_rate": 0.02,
"random_state": 42,
"early_stopping_rounds": 100
```

During the predicting stage, test data was processed with each of encoders and prediction was made via each of the models. Predictions were then ranked and summed (*ROC AUC* metric doesn't care if predictions are averaged or summed; the only importance is the order).

Results

This section contains the processed results of the experiments. If you'd like to see the raw scores for each dataset, please visit my GitHub repository — CategoricalEncodingBenchmark.

To determine the best encoder, I scaled the *ROC AUC* scores of each dataset (min-max scale) and then averaged results among the encoder. The obtained result represents the average performance score for each encoder (higher is better). The encoders performance scores for each type of validation are shown in Tables 2.1–2.3.

In order to determine the best validation strategy, I compared the top score of each dataset for each type of validation. The scores improvement (top score for a dataset

Table 2.1 Encoders performance scores - None Validation

	None Validation
HelmertEncoder	0.9517
SumEncoder	0.9434
FrequencyEncoder	0.9176
CatBoostEncoder	0.5728
TargetEncoder	0.5174
JamesSteinEncoder	0.5162
OrdinalEncoder	0.4964
WOEEncoder	0.4905
MEstimateEncoder	0.4501
BackwardDifferenceEncoder	0.4128
LeaveOneOutEncoder	0.0697

Table 2.2 Encoders performance scores - Single Validation

	Single Validation
CatBoostEncoder	0.9726
OrdinalEncoder	0.9694
HelmertEncoder	0.9558
SumEncoder	0.9434
WOEEncoder	0.9326
FrequencyEncoder	0.9315
BackwardDifferenceEncoder	0.9108
TargetEncoder	0.8915
JamesSteinEncoder	0.8555
MEstimateEncoder	0.8189
LeaveOneOutEncoder	0.0729

Table 2.3 Encoders performance scores - Double Validation

	Double Validation
JamesSteinEncoder	0.9918
CatBoostEncoder	0.9917
TargetEncoder	0.9916
LeaveOneOutEncoder	0.9909
WOEEncoder	0.9838
MEstimateEncoder	0.9686
FrequencyEncoder	0.8018

- Best encoder for each dataset and each type of validation was different. However, the non-target encoders (Helmet Encoder and Sum Encoder) dominated all other in None Validation experiment. This is quite an interesting finding because originally these types of encoders were used mostly in Linear Models.
- For the case of Single Validation, best encoders were CatBoost Encoder and Ordinal Encoder, which both had a relatively low performance score in None Validation experiment.
- Target-based encoders (James-Stein, Catboost, Target, LOO, WOE) with Double Validation showed the best performance scores from all types of validation and encoders. We might call it now the most stable and accurate approach for dealing with categorical variables.
- The performance of target-based encoders was improved by inducing the noise to train data. It could be clearly seen in the LOO Encoder — it turned out to be the worse encoder both in None and Sigle Validation with a huge gap to the second worst, but among best in Double Validation. Double Validation (or another variation of regularization) is a must for all target-based encoders.
- The results of Frequency Encoder were lowered by increasing the complexity of Validation. This is because the frequency of new categories in the test was counted on the whole test while during Single Validation it was counted on 1/5 of train dataset (5 folds in train dataset), during Double Validation — on 1/25 of train dataset (5 folds in train dataset, 5 subfolds in each fold). Thus, maximum

Table 2.4 Top score improvement (percent)

	None -> Single	Single -> Double
telecom	0.00	0.01
adult	0.02	-0.03
employee	1.98	0.39
credit	-0.01	-0.00
mortgages	0.26	-0.47
promotion	0.04	-0.20
kick	-0.05	0.06
kdd_upselling	0.10	-0.11
taxi	3.78	-0.01
poverty_A	0.74	-0.11
poverty_B	5.59	0.29
poverty_C	0.48	-0.54

Table 2.5 Encoders performance scores improvement (percent)

	None -> Single	Single -> Double
BackwardDifferenceEncoder	27.20	
CatBoostEncoder	20.10	0.40
FrequencyEncoder	0.30	-4.90
HelmertEncoder	0.20	
JamesSteinEncoder	17.70	6.30
LeaveOneOutEncoder	0.20	53.20
MEstimateEncoder	18.90	8.10
OrdinalEncoder	24.10	
SumEncoder	0.00	
TargetEncoder	19.60	4.20
WOEEncoder	23.40	1.90

Acknowledgments

I would like to thank Andrey Lukyanenko, Anton Biryukov and Daniel Potapov for fruitful discussion of the results of the work.

References

1. R library contrast coding systems for categorical variables
2. Category Encoders docs
3. Contrast Coding Systems for Categorical Variables
4. Coding schemes for categorical variables in regression, Coding systems for categorical variables in regression analysis
5. Weight of Evidence (WoE) Introductory Overview
6. James–Stein estimator, James-Stein Estimator: Definition, Formulas, Stein's Paradox in Statistics
7. A preprocessing scheme for high-cardinality categorical attributes in classification and prediction problems

9. ~~CatBoost: unbiased boosting with categorical features~~, ~~Transforming categorical~~ features to numerical features in Catboost

10. The idea of **Double Validation** was inspired by Stanislav Semenov talk on "Kaggle BNP Paribas"