Introduction

Inspired by this article and the repo, I have created the following kernel:

- Benchmarking Categorical Encoders (https://towardsdatascience.com/benchmarking-categorical-encoders-9c322bd77ee8)
- CategoricalEncodingBenchmark (https://github.com/DenisVorotyntsev/CategoricalEncodingBenchmark)

Let's see how these methods work in this dataset.

Discussion (https://www.kaggle.com/c/cat-in-the-dat/discussion/112584)

- no feature preprocessing
- Use KFold(5) for CV (+ more fold get better score)
- LR (C=0.1, solver=lbfgs)

Encoder	LB Score
TE	0.78018
WOE	0.78861
LOOE	0.79382
James-Stein	0.77843
Catboost	0.79164
One-Hot(another my kernel)	0.77973

Category-Encoders

1. Label Encoder
2. One-Hot Encoder
3. Sum Encoder
4. Helmert Encoder
5. Frequency Encoder
6. Target Encoder
7. M-Estimate Encoder
8. Weight Of Evidence Encoder
9. James-Stein Encoder
10. Leave-one-out Encoder
11. Catboost Encoder

- Validation (Benchmark)

    - single LR
    - LR with Cross Validation
- Submit

Category-Encoders

A set of scikit-learn-style transformers for encoding categorical variables into numeric by means of different techniques.

In [1]:
```
# If you want to test this on your local notebook
# http://contrib.scikit-learn.org/categorical-encoding/
# !pip install category-encoders
```

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

from category_encoders.ordinal import OrdinalEncoder
from category_encoders.woe import WOEEncoder
from category_encoders.target_encoder import TargetEncoder
from category_encoders.sum_coding import SumEncoder
from category_encoders.m_estimate import MEstimateEncoder
from category_encoders.leave_one_out import LeaveOneOutEncoder
from category_encoders.helmert import HelmertEncoder
from category_encoders.cat_boost import CatBoostEncoder
from category_encoders.james_stein import JamesSteinEncoder
from category_encoders.one_hot import OneHotEncoder

TEST = False
```

read csv and doing some preprocessing

In [3]:

```
%%time
train = pd.read_csv('/kaggle/input/cat-in-the-dat/train.csv')
test = pd.read_csv('/kaggle/input/cat-in-the-dat/test.csv')
target = train['target']
train_id = train['id']
test_id = test['id']
train.drop(['target', 'id'], axis=1, inplace=True)
test.drop('id', axis=1, inplace=True)
```

```
CPU times: user 2.06 s, sys: 292 ms, total: 2.36 s
Wall time: 2.93 s
```

In [4]:

```
feature_list = list(train.columns) # you can custumize later.
```

notation

- y and $y+$ — the total number of observations and the total number of positive observations (y=1);
- x_i, y_i — 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;
- a — a regularization hyperparameter (selected by a user), prior — an average value of the target.

1. Label Encoder (LE), Ordinary Encoder(OE)

One of the most common encoding methods.

An encoding method that converts categorical data into numbers. The code is very simple, and when you encode a specific column you can proceed as follows:

```
from sklearn.preprocessing import LabelEncoder
label = LabelEncoder()

train[column_name] = label.fit_transform(train[column_name])
```

The simple idea is to convert the same category to a number with the same value.

So the range of numbers maps from 0 to n-1 as labels.

The disadvantage is that the labels are ordered randomly (in the existing order of the data), which can add noise while assigning an unexpected order between labels. In other words, the data becomes ordinary (ordinal, ordered) data, which can lead to unintended consequences.

If you use `Category-Encoders` it will look like this code below.

In [5]:
```
%%time
LE_encoder = OrdinalEncoder(feature_list)
train_le = LE_encoder.fit_transform(train)
test_le = LE_encoder.transform(test)
```

```
CPU times: user 6.07 s, sys: 817 ms, total: 6.88 s
Wall time: 5.62 s
```

2. One-Hot Encoder (OHE, dummy encoder)

So what can you do to give values by category instead of ordering them?

If you have data with specific category values, you can create a column. If the base Label Encoder label type is N, then OHE is the way to create N columns.

Since only the row containing the content is given as 1, it is called one-hot encoding. Also called dummy encoding in the sense of creating a dummy.

In this competition:

```
traintest = pd.concat([train, test])
dummies = pd.get_dummies(traintest, columns=traintest.columns, drop_first=True,
sparse=True)
train_ohe = dummies.iloc[:train.shape[0], :]
test_ohe = dummies.iloc[train.shape[0]:, :]
train_ohe = train_ohe.sparse.to_coo().tocsr()
test_ohe = test_ohe.sparse.to_coo().tocsr()
```

If you use `Category-Encoders` it will look like this code below.

In [6]:

```
# %%time
# this method didn't work because of RAM memory.
# so we have to use pd.dummies
# OHE_encoder = OneHotEncoder(feature_list)
# train_ohe = OHE_encoder.fit_transform(train)
# test_ohe = OHE_encoder.transform(test)
```

3. Sum Encoder (Deviation Encoder, Effect Encoder)

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.

If you use `Category-Encoders` it will look like this code below.

In [7]:

```
# %%time
# this method didn't work because of RAM memory.
# SE_encoder =SumEncoder(feature_list)
# train_se = SE_encoder.fit_transform(train[feature_list], target)
# test_se = SE_encoder.transform(test[feature_list])
```

4. Helmert Encoder

Helmert Encoding 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.

This type of encoding can be useful in certain situations where levels of the categorical variable are ordered. (not this dataset)

If you use `Category-Encoders` it will look like this code below.

In [8]:

```
# %%time
# this method didn't work because of RAM memory.
# HE_encoder = HelmertEncoder(feature_list)
# train_he = HE_encoder.fit_transform(train[feature_list], target)
# test_he = HE_encoder.transform(test[feature_list])
```

5. Frequency Encoder

This method encodes by frequency.

Create a new feature with the number of categories from the training data.

I will not proceed separately in this data.

6. Target Encoder

This is a work in progress for many kernels.

The encoded category values are calculated according to the following formulas:

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

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

- mdl means **'min data in leaf'**
- a means **'smooth parameter, power of regularization'**

Target Encoder is a powerful, but it has a huuuuuge disadvantage

> **target leakage**: it uses information about the target.

To reduce the effect of target leakage,

- Increase regularization
- Add random noise to the representation of the category in train dataset (some sort of augmentation)
- Use Double Validation (using other validation)

Let's use while being careful about overfitting.

If you use `Category-Encoders` it will look like this code below.

In [9]:

```
%%time

TE_encoder = TargetEncoder()
train_te = TE_encoder.fit_transform(train[feature_list], target)
test_te = TE_encoder.transform(test[feature_list])

train_te.head()
```

```
CPU times: user 12.2 s, sys: 1.96 s, total: 14.2 s
Wall time: 10.6 s
```

Out[9]:

	bin_0	bin_1	bin_2	bin_3	bin_4	nom_0	nom_1	nom_2	nom_3
0	0	0	0	0.302537	0.290107	0.327145	0.360978	0.307162	0.242813
1	0	1	0	0.302537	0.290107	0.327145	0.290054	0.359209	0.289954
2	0	0	0	0.309384	0.290107	0.241790	0.290054	0.293085	0.289954
3	0	1	0	0.309384	0.290107	0.351052	0.290054	0.307162	0.339793
4	0	0	0	0.309384	0.333773	0.351052	0.290054	0.293085	0.339793

5 rows × 23 columns

7. M-Estimate Encoder

M-Estimate Encoder is a **simplified version of Target Encoder.** It has only one hyperparameter (Wrong Fomular but did good work?!)

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

The higher value of m results into stronger shrinking. Recommended values for m is in the range of 1 to 100.

If you use `Category-Encoders` it will look like this code below.

In [10]:
```
%%time
MEE_encoder = MEstimateEncoder()
train_mee = MEE_encoder.fit_transform(train[feature_list], target)
test_mee = MEE_encoder.transform(test[feature_list])
```

```
CPU times: user 11.6 s, sys: 1.08 s, total: 12.6 s
Wall time: 9.8 s
```

- UPDATED : error founded in libarary https://github.com/scikit-learn-contrib/category_encoders/issues/200 (https://github.com/scikit-learn-contrib/category_encoders/issues/200)

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

Thanks to @ansh422 (https://www.kaggle.com/ansh422)

8. Weight of Evidence Encoder

Weight Of Evidence is a commonly used target-based encoder in credit scoring.

It is a measure of the "strength" of a grouping for separating good and bad risk (default).

It is 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:

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

$$denominator = ln(\frac{nominator}{denominator})$$

If you use `Category-Encoders` it will look like this code below.

In [11]:
```
%%time
WOE_encoder = WOEEncoder()
train_woe = WOE_encoder.fit_transform(train[feature_list], target)
test_woe = WOE_encoder.transform(test[feature_list])
```

```
CPU times: user 11.8 s, sys: 1.25 s, total: 13.1 s
Wall time: 10.2 s
```

9. James-Stein Encoder

James-Stein Encoder is a target-based encoder.

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

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

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:

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

Seems quite fair, but James-Stein Estimator has a big disadvantage — it is defined only for normal distribution (which is not the case for any classification task).

To avoid that, 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.

If you use `Category-Encoders` it will look like this code below.

In [12]:
```
%%time
JSE_encoder = JamesSteinEncoder()
train_jse = JSE_encoder.fit_transform(train[feature_list], target)
test_jse = JSE_encoder.transform(test[feature_list])
```

```
CPU times: user 11.5 s, sys: 1.03 s, total: 12.5 s
Wall time: 9.89 s
```

10. Leave-one-out Encoder (LOO or LOOE)

Leave-one-out Encoding is another example of target-based encoders.

This encoder 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:

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

If you use `Category-Encoders` it will look like this code below.

In [13]:
```
%%time
LOOE_encoder = LeaveOneOutEncoder()
train_looe = LOOE_encoder.fit_transform(train[feature_list], target)
test_looe = LOOE_encoder.transform(test[feature_list])
```

```
CPU times: user 12.7 s, sys: 837 ms, total: 13.6 s
Wall time: 11 s
```

11. Catboost Encoder

Catboost is a recently created target-based categorical encoder.

It is intended to overcome target leakage problems inherent in LOO.

If you use `Category-Encoders` it will look like this code below.

In [14]:

```
%%time
CBE_encoder = CatBoostEncoder()
train_cbe = CBE_encoder.fit_transform(train[feature_list], target)
test_cbe = CBE_encoder.transform(test[feature_list])
```

```
CPU times: user 18.8 s, sys: 1.33 s, total: 20.1 s
Wall time: 16.7 s
```

Validation

Validation proceeds with single lr and lr with cv.

- I will add OneHotEncoder, etc later.
- More Fold get better score (my experience)
- you can try another solver and another parameter

Single LR

In [15]:

```
%%time
import gc
from sklearn.model_selection import train_test_split
from sklearn.metrics import roc_auc_score as auc
from sklearn.linear_model import LogisticRegression

encoder_list = [ OrdinalEncoder(), WOEEncoder(), TargetEncoder(), MEstima
teEncoder(), JamesSteinEncoder(), LeaveOneOutEncoder() ,CatBoostEncoder
()]

X_train, X_val, y_train, y_val = train_test_split(train, target, test_siz
e=0.2, random_state=97)

for encoder in encoder_list:
    print("Test {} : ".format(str(encoder).split('(')[0]), end=" ")
    train_enc = encoder.fit_transform(X_train[feature_list], y_train)
    #test_enc = encoder.transform(test[feature_list])
    val_enc = encoder.transform(X_val[feature_list])
    lr = LogisticRegression(C=0.1, solver="lbfgs", max_iter=1000)
    lr.fit(train_enc, y_train)
    lr_pred = lr.predict_proba(val_enc)[:, 1]
    score = auc(y_val, lr_pred)
    print("score: ", score)
    del train_enc
    del val_enc
    gc.collect()
```

```
Test OrdinalEncoder :


/opt/conda/lib/python3.6/site-packages/sklearn/linear_model/logistic.p
y:947: ConvergenceWarning: lbfgs failed to converge. Increase the numb
er of iterations.
  "of iterations.", ConvergenceWarning)


score:  0.6058952245541614
Test WOEEncoder :  score:  0.7814115175120097
Test TargetEncoder :  score:  0.7790232741835907
Test MEstimateEncoder :  score:  0.7792025608680453
Test JamesSteinEncoder :  score:  0.7719623937439537
Test LeaveOneOutEncoder :  score:  0.79576549204064
Test CatBoostEncoder :  score:  0.7917094405644212
CPU times: user 5min 8s, sys: 2.29 s, total: 5min 10s
Wall time: 3min 21s
```

LR with CrossValidation

In [16]:

```
%%time
from sklearn.model_selection import KFold
import numpy as np

# CV function original : @Peter Hurford : Why Not Logistic Regression? http
s://www.kaggle.com/peterhurford/why-not-logistic-regression

def run_cv_model(train, test, target, model_fn, params={}, label='mode
l'):
    kf = KFold(n_splits=5)
    fold_splits = kf.split(train, target)

    cv_scores = []
    pred_full_test = 0
    pred_train = np.zeros((train.shape[0]))
    i = 1
    for dev_index, val_index in fold_splits:
        print('Started {} fold {}/5'.format(label, i))
        dev_X, val_X = train.iloc[dev_index], train.iloc[val_index]
        dev_y, val_y = target[dev_index], target[val_index]
        pred_val_y, pred_test_y = model_fn(dev_X, dev_y, val_X, val_y, te
st, params)
        pred_full_test = pred_full_test + pred_test_y
        pred_train[val_index] = pred_val_y
        cv_score = auc(val_y, pred_val_y)
        cv_scores.append(cv_score)
        print(label + ' cv score {}: {}'.format(i, cv_score))
        i += 1

    print('{} cv scores : {}'.format(label, cv_scores))
    print('{} cv mean score : {}'.format(label, np.mean(cv_scores)))
    print('{} cv std score : {}'.format(label, np.std(cv_scores)))
    pred_full_test = pred_full_test / 5.0
    results = {'label': label, 'train': pred_train, 'test': pred_full_tes
t, 'cv': cv_scores}
    return results


def runLR(train_X, train_y, test_X, test_y, test_X2, params):
    model = LogisticRegression(**params)
```

```
    model.fit(train_X, train_y)
    pred_test_y = model.predict_proba(test_X)[:, 1]
    pred_test_y2 = model.predict_proba(test_X2)[:, 1]
    return pred_test_y, pred_test_y2
```

Show hidden output

In [17]:

```
if TEST:

    lr_params = {'solver': 'lbfgs', 'C': 0.1}

    results = list()

    for encoder in  [ OrdinalEncoder(), WOEEncoder(), TargetEncoder(), ME
stimateEncoder(), JamesSteinEncoder(), LeaveOneOutEncoder() ,CatBoostEnco
der()]:
        train_enc = encoder.fit_transform(train[feature_list], target)
        test_enc = encoder.transform(test[feature_list])
        result = run_cv_model(train_enc, test_enc, target, runLR, lr_para
ms, str(encoder).split('(')[0])
        results.append(result)
    results = pd.DataFrame(results)
    results['cv_mean'] = results['cv'].apply(lambda l : np.mean(l))
    results['cv_std'] = results['cv'].apply(lambda l : np.std(l))
    results[['label','cv_mean','cv_std']].head(8)
```

Submit

Even CVs did not solve the target based encoder's overfit problem.

In [18]:

```
if TEST:
    for idx, label in enumerate(results['label']):
        sub_df = pd.DataFrame({'id': test_id, 'target' : results.iloc[id
x]['test']})
        sub_df.to_csv("LR_{}.csv".format(label), index=False)
```

In [ ]: