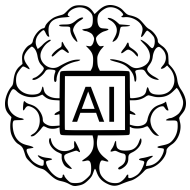
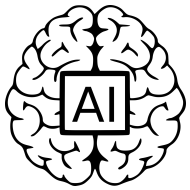


Encoding Of Categorical Variables

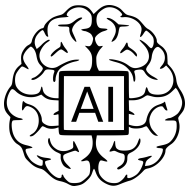
Michael Matějů

AI Squad

July 30, 2023

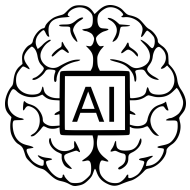


Outline



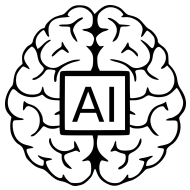
Motivation

- MRMR Feature Selection Algorithm implemented by Smazzanti
- For categorical encoding he used three target based algorithms
- And inspite of all common sense it works in Kaggle Challanges
- There is no free lunch.
- Regularization is a must for target-based encoders.



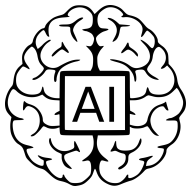
Motivation

- MRMR Feature Selection Algorithm implemented by Smazzanti
- For categorical encoding he used three target based algorithms
- And inspite of all common sense it works in Kaggle Challanges
- There is no free lunch.
- Regularization is a must for target-based encoders.



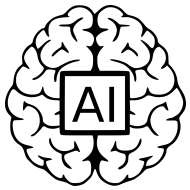
Motivation

- MRMR Feature Selection Algorithm implemented by Smazzanti
- For categorical encoding he used three target based algorithms
- And inspite of all common sense it works in Kaggle Challanges
- There is no free lunch.
- Regularization is a must for target-based encoders.



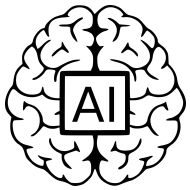
Motivation

- MRMR Feature Selection Algorithm implemented by Smazzanti
- For categorical encoding he used three target based algorithms
- And inspite of all common sense it works in Kaggle Challanges
- There is no free lunch.
- Regularization is a must for target-based encoders.



Motivation

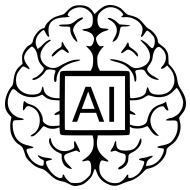
- MRMR Feature Selection Algorithm implemented by Smazzanti
- For categorical encoding he used three target based algorithms
- And inspite of all common sense it works in Kaggle Challanges
- There is no free lunch.
- Regularization is a must for target-based encoders.



Toy Dataset

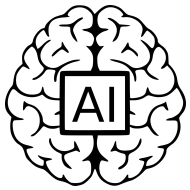
	category	target
0	A	0
1	A	1
2	A	0
3	A	1
4	B	0
5	B	1
6	B	0
7	C	1
8	C	0
9	D	1

Table: Toy Dataset



Types

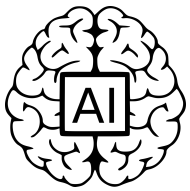
- Target-Based Encoders
- The Rest



Label / Ordinary Encoder

- One of the most common algorithms
- Encodes into N integers

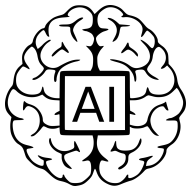
```
LE_encoder = OrdinalEncoder(feature_list)
train_le = LE_encoder.fit_transform(train)
test_le = LE_encoder.transform(test)
```



Label / Ordinary Encoder

- One of the most common algorithms
- Encodes into N integers

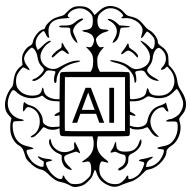
```
LE_encoder = OrdinalEncoder(feature_list)
train_le = LE_encoder.fit_transform(train)
test_le = LE_encoder.transform(test)
```



One Hot Encoder

- One of the most common algorithms
- Encodes columns with N categories into N-1 boolean columns

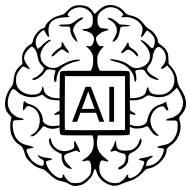
```
traintest = pd.concat([train, test])
dummies = pd.get_dummies(traintest,
    columns=traintest.columns, drop_first=True,
    sparse=True)
train_ohe = dummies.iloc[:train.shape[0], :]
test_ohe = dummies.iloc[train.shape[0]:, :]
train_ohe = train_ohe.sparse.to_coo().tocsr()
test_ohe = test_ohe.sparse.to_coo().tocsr()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



One Hot Encoder

- One of the most common algorithms
- Encodes columns with N categories into N-1 boolean columns

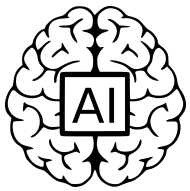
```
traintest = pd.concat([train, test])
dummies = pd.get_dummies(traintest,
    columns=traintest.columns, drop_first=True,
    sparse=True)
train_ohe = dummies.iloc[:train.shape[0], :]
test_ohe = dummies.iloc[train.shape[0]:, :]
train_ohe = train_ohe.sparse.to_coo().tocsr()
test_ohe = test_ohe.sparse.to_coo().tocsr()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



Sum / Deviation Encoder

- Similar to One Hot Encoder
- Difference is in interpretation of LR coefficients.

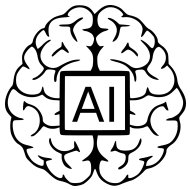
```
# this method didn't work because of low RAM memory.
# SE_encoder =SumEncoder(feature_list)
# train_se =
    SE_encoder.fit_transform(train[feature_list],
    target)
# test_se = SE_encoder.transform(test[feature_list])
```



Sum / Deviation Encoder

- Similar to One Hot Encoder
- Difference is in interpretation of LR coefficients.

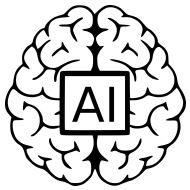
```
# this method didn't work because of low RAM memory.
# SE_encoder =SumEncoder(feature_list)
# train_se =
    SE_encoder.fit_transform(train[feature_list],
    target)
# test_se = SE_encoder.transform(test[feature_list])
```



Helmert Encoder

- Another common algorithm for regression of ordered categorical data
- Compares each level of a categorical variable to the mean of the subsequent levels (so-called forward Helmert) or previous levels (so-called reverse Helmert).

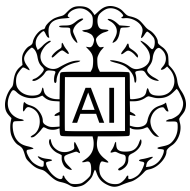
```
# this method didn't work because of RAM memory.
# HE_encoder = HelmertEncoder(feature_list)
# train_he =
    HE_encoder.fit_transform(train[feature_list],
    target)
# test_he = HE_encoder.transform(test[feature_list])
```



Helmert Encoder

- Another common algorithm for regression of ordered categorical data
- Compares each level of a categorical variable to the mean of the subsequent levels (so-called forward Helmert) or previous levels (so-called reverse Helmert).

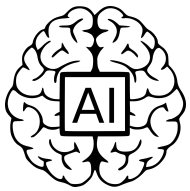
```
# this method didn't work because of RAM memory.
# HE_encoder = HelmertEncoder(feature_list)
# train_he =
    HE_encoder.fit_transform(train[feature_list],
    target)
# test_he = HE_encoder.transform(test[feature_list])
```



Helmert Encoder

- Another common algorithm for regression of ordered categorical data
- Compares each level of a categorical variable to the mean of the subsequent levels (so-called forward Helmert) or previous levels (so-called reverse Helmert).

```
# 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])
```

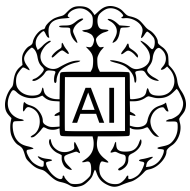


Frequency Encoder

- Counts the number of a category's occurrences in the dataset
- Little bit tricky to use (for different num of occurencies in train/test set)

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

Table: Toy Dataset

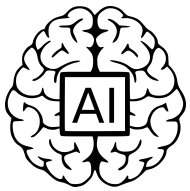


Frequency Encoder

- Counts the number of a category's occurrences in the dataset
- Little bit tricky to use (for different num of occurencies in train/test set)

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

Table: Toy Dataset

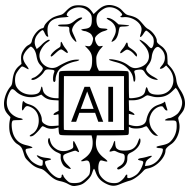


Frequency Encoder

- Counts the number of a category's occurrences in the dataset
- Little bit tricky to use (for different num of occurencies in train/test set)

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

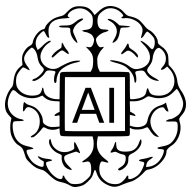
Table: Toy Dataset



Target Encoder / Mean 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.
- Main idea:
  1. Select a categorical variable you would like to transform.
  2. Group by the categorical variable and obtain aggregated sum over the "Target" variable. (total number of 1's for each category in "category" column)
  3. Group by the categorical variable and obtain aggregated count over "Target" variable
  4. Divide the step 2 / step 3 results and join it back with the train.

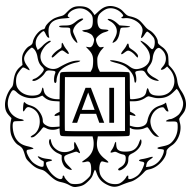
```
mean_encode = df.groupby('category')['target'].mean()
df.loc[:, 'category_representation'] =
    df['category'].map(mean_encode)
```



Target Encoder / Mean 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.
- Main idea:
  1. Select a categorical variable you would like to transform.
  2. Group by the categorical variable and obtain aggregated sum over the "Target" variable. (total number of 1's for each category in "category" column)
  3. Group by the categorical variable and obtain aggregated count over "Target" variable
  4. Divide the step 2 / step 3 results and join it back with the train.

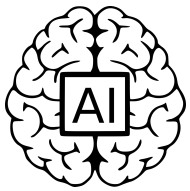
```
mean_encode = df.groupby('category')['target'].mean()
df.loc[:, 'category_representation'] =
    df['category'].map(mean_encode)
```



Target Encoder / Mean 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.
- Main idea:
  1. Select a categorical variable you would like to transform.
  2. Group by the categorical variable and obtain aggregated sum over the "Target" variable. (total number of 1's for each category in "category" column)
  3. Group by the categorical variable and obtain aggregated count over "Target" variable
  4. Divide the step 2 / step 3 results and join it back with the train.

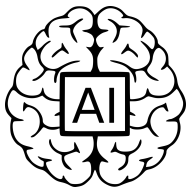
```
mean_encode = df.groupby('category')['target'].mean()
df.loc[:, 'category_representation'] =
    df['category'].map(mean_encode)
```



Target Encoder / Mean 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.
- Main idea:
  1. Select a categorical variable you would like to transform.
  2. Group by the categorical variable and obtain aggregated sum over the "Target" variable. (total number of 1's for each category in "category" column)
  3. Group by the categorical variable and obtain aggregated count over "Target" variable
  4. Divide the step 2 / step 3 results and join it back with the train.

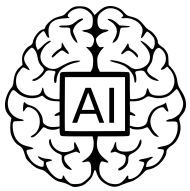
```
mean_encode = df.groupby('category')['target'].mean()
df.loc[:, 'category_representation'] =
    df['category'].map(mean_encode)
```



Target Encoder / Mean 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.
- Main idea:
  1. Select a categorical variable you would like to transform.
  2. Group by the categorical variable and obtain aggregated sum over the "Target" variable. (total number of 1's for each category in "category" column)
  3. Group by the categorical variable and obtain aggregated count over "Target" variable
  4. Divide the step 2 / step 3 results and join it back with the train.

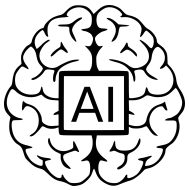
```
mean_encode = df.groupby('category')['target'].mean()
df.loc[:, 'category_representation'] =
    df['category'].map(mean_encode)
```



Target Encoder / Mean 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.
- Main idea:
  1. Select a categorical variable you would like to transform.
  2. Group by the categorical variable and obtain aggregated sum over the "Target" variable. (total number of 1's for each category in "category" column)
  3. Group by the categorical variable and obtain aggregated count over "Target" variable
  4. Divide the step 2 / step 3 results and join it back with the train.

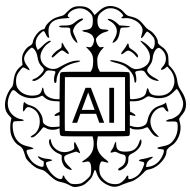
```
mean_encode = df.groupby('category')['target'].mean()
df.loc[:, 'category_representation'] =
    df['category'].map(mean_encode)
```



Target Encoder / Mean 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.
- Main idea:
  1. Select a categorical variable you would like to transform.
  2. Group by the categorical variable and obtain aggregated sum over the "Target" variable. (total number of 1's for each category in "category" column)
  3. Group by the categorical variable and obtain aggregated count over "Target" variable
  4. Divide the step 2 / step 3 results and join it back with the train.

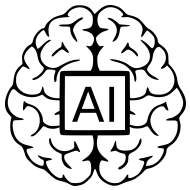
```
mean_encode = df.groupby('category')['target'].mean()
df.loc[:, 'category_representation'] =
    df['category'].map(mean_encode)
```



Target Encoder / Mean 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.
- Main idea:
  1. Select a categorical variable you would like to transform.
  2. Group by the categorical variable and obtain aggregated sum over the "Target" variable. (total number of 1's for each category in "category" column)
  3. Group by the categorical variable and obtain aggregated count over "Target" variable
  4. Divide the step 2 / step 3 results and join it back with the train.

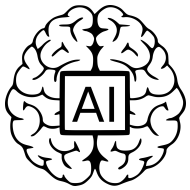
```
mean_encode = df.groupby('category')['target'].mean()
df.loc[:, 'category_representation'] =
    df['category'].map(mean_encode)
```



Target Encoder / Mean Encoding

	category	category representation	target
0	A	0.5	0
1	A	0.5	1
2	A	0.5	0
3	A	0.5	1
4	B	0.33	0
5	B	0.33	1
6	B	0.33	0
7	C	0.5	1
8	C	0.5	0
9	D	1	1

Table: Toy Dataset Target Encoding



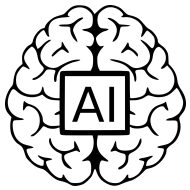
Target Encoder / Mean Encoding

Issues:

- Target Leakage
- Mean is good but not perfect. We train the models on fraction of data. We could have either luck or bad luck in training the encoder on the data. We might even over -fit data.

To reduce the effect of leakage:

- Increase regularization
- Add random noise to representation of the category in train set
- Use Double Validation



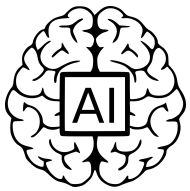
Target Encoder / Mean Encoding

Issues:

- Target Leakage
- Mean is good but not perfect. We train the models on fraction of data. We could have either luck or bad luck in training the encoder on the data. We might even over -fit data.

To reduce the effect of leakage:

- Increase regularization
- Add random noise to representation of the category in train set
- Use Double Validation



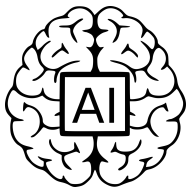
Target Encoder / Mean Encoding

Issues:

- Target Leakage
- Mean is good but not perfect. We train the models on fraction of data. We could have either luck or bad luck in training the encoder on the data. We might even over -fit data.

To reduce the effect of leakage:

- Increase regularization
- Add random noise to representation of the category in train set
- Use Double Validation



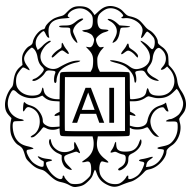
Target Encoder / Mean Encoding

Issues:

- Target Leakage
- Mean is good but not perfect. We train the models on fraction of data. We could have either luck or bad luck in training the encoder on the data. We might even over -fit data.

To reduce the effect of leakage:

- Increase regularization
- Add random noise to representation of the category in train set
- Use Double Validation



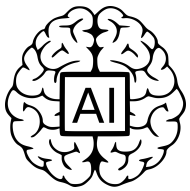
Target Encoder / Mean Encoding

Issues:

- Target Leakage
- Mean is good but not perfect. We train the models on fraction of data. We could have either luck or bad luck in training the encoder on the data. We might even over -fit data.

To reduce the effect of leakage:

- Increase regularization
- Add random noise to representation of the category in train set
- Use Double Validation



Target Encoder / Mean Encoding

Target Encoding with prior smoothing:

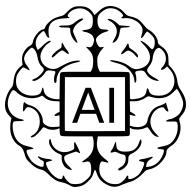
$$encoding = \alpha \cdot p(t = 1|x = c_i) + (1 - \alpha) \cdot p(t = 1),$$

where

$$\alpha = \frac{1}{1 + exp(-\frac{n-k}{f})},$$

where

- k means 'min samples leaf'
- f means 'smooth parameter, power of regularization'
- n means the number of observations for a given value of a categorical column;

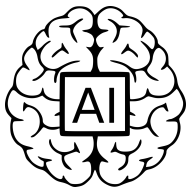


Target Encoder / Mean Encoding


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

	bin_0	bin_1	bin_2	bin_3	bin_4	nom_0	nom_1	nom_2	nom_3	nom_4	...
0	0	0	0	0.302537	0.290107	0.327145	0.360978	0.307162	0.242813	0.237743	...
1	0	1	0	0.302537	0.290107	0.327145	0.290054	0.359209	0.289954	0.304164	...
2	0	0	0	0.309384	0.290107	0.241790	0.290054	0.293085	0.289954	0.353951	...
3	0	1	0	0.309384	0.290107	0.351052	0.290054	0.307162	0.339793	0.329472	...
4	0	0	0	0.309384	0.333773	0.351052	0.290054	0.293085	0.339793	0.329472	...

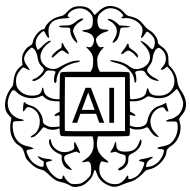
5 rows × 23 columns



Leave-one-out Encoder

- Another target based encoder. Similar to default target encoder with the slight difference that to current observation is leave out of calculation.
- $\hat{x}_i^k = \frac{\sum_{j \neq i}(y_j \cdot (x_j == k))}{\sum_{j \neq i} x_j == k}$
- To prevend the target leakage the formula is augmented by randomness and regularization:
  $\hat{x}_i^k = \frac{\sum_{j \neq i}(y_j \cdot (x_j == k))}{\sum_j (x_j == k) - 1 + R} \cdot (1 + \epsilon_i)$
- where R is regularization factor and $\epsilon \approx N(0, \sigma^2)$.
- Problem is shift in encoding of train and test categories.

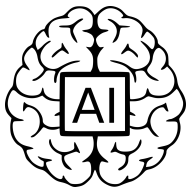
```
LOOE_encoder = LeaveOneOutEncoder()
train_looe =
    LOOE_encoder.fit_transform(train[feature_list],
    target)
test_looe = LOOE_encoder.transform(test[feature_list])
```



Leave-one-out Encoder

- Another target based encoder. Similar to default target encoder with the slight difference that to current observation is leave out of calculation.
- $\hat{x}_i^k = \frac{\sum_{j \neq i}(y_j \cdot (x_j == k))}{\sum_{j \neq i} x_j == k}$
- To prevend the target leakage the formula is augmented by randomness and regularization:
  $\hat{x}_i^k = \frac{\sum_{j \neq i}(y_j \cdot (x_j == k))}{\sum_j (x_j == k) - 1 + R} \cdot (1 + \epsilon_i)$
- where R is regularization factor and $\epsilon \approx N(0, \sigma^2)$.
- Problem is shift in encoding of train and test categories.

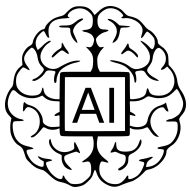
```
LOOE_encoder = LeaveOneOutEncoder()
train_looe =
    LOOE_encoder.fit_transform(train[feature_list],
    target)
test_looe = LOOE_encoder.transform(test[feature_list])
```



Leave-one-out Encoder

- Another target based encoder. Similar to default target encoder with the slight difference that to current observation is leave out of calculation.
- $\hat{x}_i^k = \frac{\sum_{j \neq i}(y_j \cdot (x_j == k))}{\sum_{j \neq i} x_j == k}$
- To prevend the target leakage the formula is augmented by randomness and regularization:
  $\hat{x}_i^k = \frac{\sum_{j \neq i}(y_j \cdot (x_j == k))}{\sum_j (x_j == k) - 1 + R} \cdot (1 + \epsilon_i)$
- where R is regularization factor and $\epsilon \approx N(0, \sigma^2)$.
- Problem is shift in encoding of train and test categories.

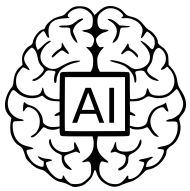
```
LOOE_encoder = LeaveOneOutEncoder()
train_looe =
    LOOE_encoder.fit_transform(train[feature_list],
    target)
test_looe = LOOE_encoder.transform(test[feature_list])
```



Leave-one-out Encoder

- Another target based encoder. Similar to default target encoder with the slight difference that to current observation is leave out of calculation.
- $\hat{x}_i^k = \frac{\sum_{j \neq i}(y_j \cdot (x_j == k))}{\sum_{j \neq i} x_j == k}$
- To prevend the target leakage the formula is augmented by randomness and regularization:
  $\hat{x}_i^k = \frac{\sum_{j \neq i}(y_j \cdot (x_j == k))}{\sum_j (x_j == k) - 1 + R} \cdot (1 + \epsilon_i)$
- where R is regularization factor and $\epsilon \approx N(0, \sigma^2)$.
- Problem is shift in encoding of train and test categories.

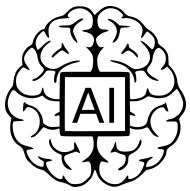
```
LOOE_encoder = LeaveOneOutEncoder()
train_looe =
    LOOE_encoder.fit_transform(train[feature_list],
    target)
test_looe = LOOE_encoder.transform(test[feature_list])
```



Leave-one-out Encoder

- Another target based encoder. Similar to default target encoder with the slight difference that to current observation is leave out of calculation.
- $\hat{x}_i^k = \frac{\sum_{j \neq i}(y_j \cdot (x_j == k))}{\sum_{j \neq i} x_j == k}$
- To prevend the target leakage the formula is augmented by randomness and regularization:
  $\hat{x}_i^k = \frac{\sum_{j \neq i}(y_j \cdot (x_j == k))}{\sum_j (x_j == k) - 1 + R} \cdot (1 + \epsilon_i)$
- where R is regularization factor and $\epsilon \approx N(0, \sigma^2)$.
- Problem is shift in encoding of train and test categories.

```
LOOE_encoder = LeaveOneOutEncoder()
train_looe =
    LOOE_encoder.fit_transform(train[feature_list],
    target)
test_looe = LOOE_encoder.transform(test[feature_list])
```



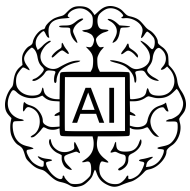
M-Estimate Encoder

- Handles outliers assigning weights to each category based on its deviation from overall class frequency.

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

where m is the smoothing hyperparameter.

```
MEE_encoder = MEstimateEncoder()
train_mee =
    MEE_encoder.fit_transform(train[feature_list],
    target)
test_mee = MEE_encoder.transform(test[feature_list])
```

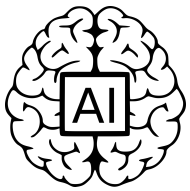


Weight of Evidence Encoder

- Commonly used in credit scoring. It is the measure of "strength" of grouping for separating good and bad risk.
  - a = Distribution of Good Credit Outcomes
  - b = Distribution of Bad Credit Outcomes
  - WoE = ln(a/b)

In practice, those formulas might lead to target leakage and therefore the WoE is calculated with a smoothing hyperparameter.

```
WOE_encoder = WOEEncoder()
train_woe =
    WOE_encoder.fit_transform(train[feature_list],
    target)
test_woe = WOE_encoder.transform(test[feature_list])
```

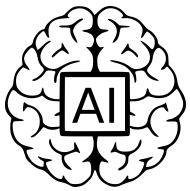


Weight of Evidence Encoder

- Commonly used in credit scoring. It is the measure of "strength" of grouping for separating good and bad risk.
- a = Distribution of Good Credit Outcomes
- b = Distribution of Bad Credit Outcomes
- WoE = ln(a/b)

In practice, those formulas might lead to target leakage and therefore the WoE is calculated with a smoothing hyperparameter.

```
WOE_encoder = WOEEncoder()
train_woe =
    WOE_encoder.fit_transform(train[feature_list],
    target)
test_woe = WOE_encoder.transform(test[feature_list])
```

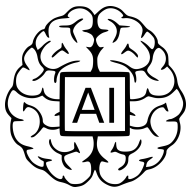


Weight of Evidence Encoder

- Commonly used in credit scoring. It is the measure of "strength" of grouping for separating good and bad risk.
- a = Distribution of Good Credit Outcomes
- b = Distribution of Bad Credit Outcomes
- WoE = ln(a/b)

In practice, those formulas might lead to target leakage and therefore the WoE is calculated with a smoothing hyperparameter.

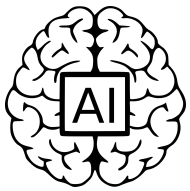
```
WOE_encoder = WOEEncoder()
train_woe =
    WOE_encoder.fit_transform(train[feature_list],
    target)
test_woe = WOE_encoder.transform(test[feature_list])
```



Probability Ratio Encoder

- Similar to Weight Of Evidence(WoE), with the only difference that the ratio of good and bad probability is used.
- For each label, we calculate the mean of target=1, i.e. the probability of being 1 (P(1)).
- Also we calculate the probability of the target=0 (P(0)).
- And then, we calculate the ratio P(1)/P(0) and replace the labels with that ratio.

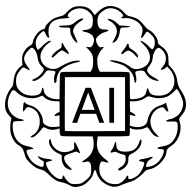
```
pr_df = df.groupby("month")["target"].mean()
pr_df = pd.DataFrame(pr_df)
pr_df = pr_df.rename(columns = {"target" : "good"})
pr_df["bad"] = 1-pr_df.good
pr_df["bad"] = np.where(pr_df["bad"] == 0, 0.000001,
    pr_df["bad"])
pr_df["PR"] = pr_df.good / pr_df.bad
df.loc[:, "PR_Encode"] = df["month"].map(pr_df["PR"])
```



Probability Ratio Encoder

- Similar to Weight Of Evidence(WoE), with the only difference that the ratio of good and bad probability is used.
- For each label, we calculate the mean of target=1, i.e. the probability of being 1 (P(1)).
- Also we calculate the probability of the target=0 (P(0)).
- And then, we calculate the ratio P(1)/P(0) and replace the labels with that ratio.

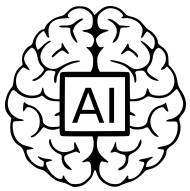
```
pr_df = df.groupby("month")["target"].mean()
pr_df = pd.DataFrame(pr_df)
pr_df = pr_df.rename(columns = {"target" : "good"})
pr_df["bad"] = 1-pr_df.good
pr_df["bad"] = np.where(pr_df["bad"] == 0, 0.000001,
    pr_df["bad"])
pr_df["PR"] = pr_df.good / pr_df.bad
df.loc[:, "PR_Encode"] = df["month"].map(pr_df["PR"])
```



Probability Ratio Encoder

- Similar to Weight Of Evidence(WoE), with the only difference that the ratio of good and bad probability is used.
- For each label, we calculate the mean of target=1, i.e. the probability of being 1 (P(1)).
- Also we calculate the probability of the target=0 (P(0)).
- And then, we calculate the ratio P(1)/P(0) and replace the labels with that ratio.

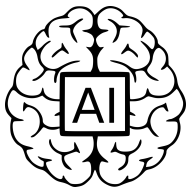
```
pr_df = df.groupby("month")["target"].mean()
pr_df = pd.DataFrame(pr_df)
pr_df = pr_df.rename(columns = {"target" : "good"})
pr_df["bad"] = 1-pr_df.good
pr_df["bad"] = np.where(pr_df["bad"] == 0, 0.000001,
    pr_df["bad"])
pr_df["PR"] = pr_df.good / pr_df.bad
df.loc[:, "PR_Encode"] = df["month"].map(pr_df["PR"])
```



Probability Ratio Encoder

- Similar to Weight Of Evidence(WoE), with the only difference that the ratio of good and bad probability is used.
- For each label, we calculate the mean of target=1, i.e. the probability of being 1 (P(1)).
- Also we calculate the probability of the target=0 (P(0)).
- And then, we calculate the ratio P(1)/P(0) and replace the labels with that ratio.

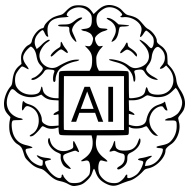
```
pr_df = df.groupby("month")["target"].mean()
pr_df = pd.DataFrame(pr_df)
pr_df = pr_df.rename(columns = {"target" : "good"})
pr_df["bad"] = 1-pr_df.good
pr_df["bad"] = np.where(pr_df["bad"] == 0, 0.000001,
    pr_df["bad"])
pr_df["PR"] = pr_df.good / pr_df.bad
df.loc[:, "PR_Encode"] = df["month"].map(pr_df["PR"])
```



Probability Ratio Encoder

- Similar to Weight Of Evidence(WoE), with the only difference that the ratio of good and bad probability is used.
- For each label, we calculate the mean of target=1, i.e. the probability of being 1 (P(1)).
- Also we calculate the probability of the target=0 (P(0)).
- And then, we calculate the ratio P(1)/P(0) and replace the labels with that ratio.

```
pr_df = df.groupby("month")["target"].mean()
pr_df = pd.DataFrame(pr_df)
pr_df = pr_df.rename(columns = {"target" : "good"})
pr_df["bad"] = 1-pr_df.good
pr_df["bad"] = np.where(pr_df["bad"] == 0, 0.000001,
    pr_df["bad"])
pr_df["PR"] = pr_df.good / pr_df.bad
df.loc[:, "PR_Encode"] = df["month"].map(pr_df["PR"])
```



James-Stein Encoder

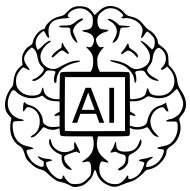
- Estimation of the mean target for category k could be calculated by:
- $\hat{x}^k = (1 - B) \cdot \frac{n^+}{n} + B \cdot \frac{y^+}{y}$

B is a hyperparameter but J-S found a formula to calculate it:

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

It is defined only for normal distribution. Not the case for any classification task. There is workaround for it - using log-odds ratios instead (default option of the transformer).

```
JSE_encoder = JamesSteinEncoder()
train_jse =
    JSE_encoder.fit_transform(train[feature_list],
    target)
test_jse = JSE_encoder.transform(test[feature_list])
```



James-Stein Encoder

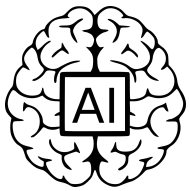
- Estimation of the mean target for category k could be calculated by:
- $\hat{x}^k = (1 - B) \cdot \frac{n^+}{n} + B \cdot \frac{y^+}{y}$

B is a hyperparameter but J-S found a formula to calculate it:

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

It is defined only for normal distribution. Not the case for any classification task. There is workaround for it - using log-odds ratios instead (default option of the transformer).

```
JSE_encoder = JamesSteinEncoder()
train_jse =
    JSE_encoder.fit_transform(train[feature_list],
    target)
test_jse = JSE_encoder.transform(test[feature_list])
```



James-Stein Encoder

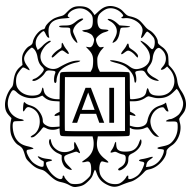
- Estimation of the mean target for category k could be calculated by:
- $\hat{x}^k = (1 - B) \cdot \frac{n^+}{n} + B \cdot \frac{y^+}{y}$

B is a hyperparameter but J-S found a formula to calculate it:

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

It is defined only for normal distribution. Not the case for any classification task. There is workaround for it - using log-odds ratios instead (default option of the transformer).

```
JSE_encoder = JamesSteinEncoder()
train_jse =
    JSE_encoder.fit_transform(train[feature_list],
    target)
test_jse = JSE_encoder.transform(test[feature_list])
```



James-Stein Encoder

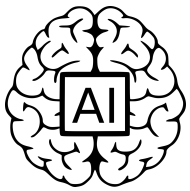
- Estimation of the mean target for category k could be calculated by:
- $\hat{x}^k = (1 - B) \cdot \frac{n^+}{n} + B \cdot \frac{y^+}{y}$

B is a hyperparameter but J-S found a formula to calculate it:

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

It is defined only for normal distribution. Not the case for any classification task. There is workaround for it - using log-odds ratios instead (default option of the transformer).

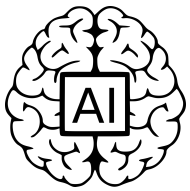
```
JSE_encoder = JamesSteinEncoder()
train_jse =
    JSE_encoder.fit_transform(train[feature_list],
    target)
test_jse = JSE_encoder.transform(test[feature_list])
```



Catboost Encoder

- Catboost is a recently created target-based categorical encoder.
- Catboost overcomes the target leakage by introducing time into dataset - the order of the observations.
- $\hat{x}_i^k = \frac{\sum_{j=0}^{j\leq i}(y_j \cdot (x_j == k)) - y_i + prior}{\sum_{j=0}^{j\leq i}(x_j == k) + 1}$
- To prevent the over-fitting, the process is repeated several times on shuffled dataset and results are averaged.
- Catboost "on-the-fly" encoding is one of the core advantages of CatBoost.

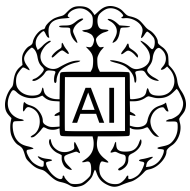
```
CBE_encoder = CatBoostEncoder()
train_cbe =
    CBE_encoder.fit_transform(train[feature_list],
    target)
test_cbe = CBE_encoder.transform(test[feature_list])
```



Catboost Encoder

- Catboost is a recently created target-based categorical encoder.
- Catboost overcomes the target leakage by introducing time into dataset - the order of the observations.
- $\hat{x}_i^k = \frac{\sum_{j=0}^{j\leq i}(y_j \cdot (x_j == k)) - y_i + prior}{\sum_{j=0}^{j\leq i}(x_j == k) + 1}$
- To prevent the over-fitting, the process is repeated several times on shuffled dataset and results are averaged.
- Catboost "on-the-fly" encoding is one of the core advantages of CatBoost.

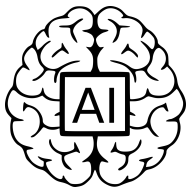
```
CBE_encoder = CatBoostEncoder()
train_cbe =
    CBE_encoder.fit_transform(train[feature_list],
    target)
test_cbe = CBE_encoder.transform(test[feature_list])
```



Catboost Encoder

- Catboost is a recently created target-based categorical encoder.
- Catboost overcomes the target leakage by introducing time into dataset - the order of the observations.
- $\hat{x}_i^k = \frac{\sum_{j=0}^{j\leq i}(y_j \cdot (x_j == k)) - y_i + prior}{\sum_{j=0}^{j\leq i}(x_j == k) + 1}$
- To prevent the over-fitting, the process is repeated several times on shuffled dataset and results are averaged.
- Catboost "on-the-fly" encoding is one of the core advantages of CatBoost.

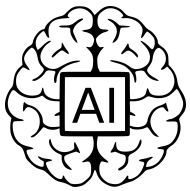
```
CBE_encoder = CatBoostEncoder()
train_cbe =
    CBE_encoder.fit_transform(train[feature_list],
    target)
test_cbe = CBE_encoder.transform(test[feature_list])
```



Catboost Encoder

- Catboost is a recently created target-based categorical encoder.
- Catboost overcomes the target leakage by introducing time into dataset - the order of the observations.
- $\hat{x}_i^k = \frac{\sum_{j=0}^{j\leq i}(y_j \cdot (x_j == k)) - y_i + prior}{\sum_{j=0}^{j\leq i}(x_j == k) + 1}$
- To prevent the over-fitting, the process is repeated several times on shuffled dataset and results are averaged.
- Catboost "on-the-fly" encoding is one of the core advantages of CatBoost.

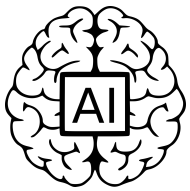
```
CBE_encoder = CatBoostEncoder()
train_cbe =
    CBE_encoder.fit_transform(train[feature_list],
    target)
test_cbe = CBE_encoder.transform(test[feature_list])
```



Catboost Encoder

- Catboost is a recently created target-based categorical encoder.
- Catboost overcomes the target leakage by introducing time into dataset - the order of the observations.
- $\hat{x}_i^k = \frac{\sum_{j=0}^{j\leq i}(y_j \cdot (x_j == k)) - y_i + prior}{\sum_{j=0}^{j\leq i}(x_j == k) + 1}$
- To prevent the over-fitting, the process is repeated several times on shuffled dataset and results are averaged.
- Catboost "on-the-fly" encoding is one of the core advantages of CatBoost.

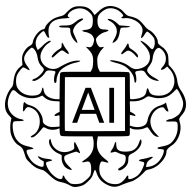
```
CBE_encoder = CatBoostEncoder()
train_cbe =
    CBE_encoder.fit_transform(train[feature_list],
    target)
test_cbe = CBE_encoder.transform(test[feature_list])
```



Catboost Encoder

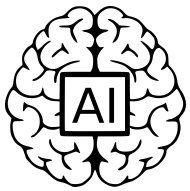
- Catboost is a recently created target-based categorical encoder.
- Catboost overcomes the target leakage by introducing time into dataset - the order of the observations.
- $\hat{x}_i^k = \frac{\sum_{j=0}^{j\leq i}(y_j \cdot (x_j == k)) - y_i + prior}{\sum_{j=0}^{j\leq i}(x_j == k) + 1}$
- To prevent the over-fitting, the process is repeated several times on shuffled dataset and results are averaged.
- Catboost "on-the-fly" encoding is one of the core advantages of CatBoost.

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



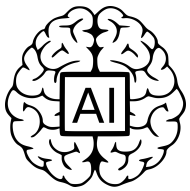
Hashing Encoder

- Converts categorical variables to a higher dimensional space of integers.
- Maintains the distance between two vectors of categorical variables
- The number of dimensions will be far less than One Hot Encoding
- Data are transformed into fewer features - information loss.
- Different categorical values could be represented by the same Hash values.
- Many Kaggle competitors use Hash Encoding to win the competition, so it is worth a try.
- Used in production when a category changes very frequently.



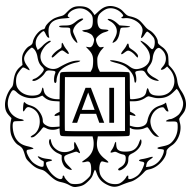
Hashing Encoder

- Converts categorical variables to a higher dimensional space of integers.
- Maintains the distance between two vectors of categorical variables
- The number of dimensions will be far less than One Hot Encoding
- Data are transformed into fewer features - information loss.
- Different categorical values could be represented by the same Hash values.
- Many Kaggle competitors use Hash Encoding to win the competition, so it is worth a try.
- Used in production when a category changes very frequently.



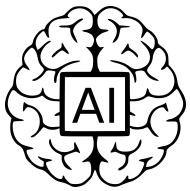
Hashing Encoder

- Converts categorical variables to a higher dimensional space of integers.
- Maintains the distance between two vectors of categorical variables
- The number of dimensions will be far less than One Hot Encoding
- Data are transformed into fewer features - information loss.
- Different categorical values could be represented by the same Hash values.
- Many Kaggle competitors use Hash Encoding to win the competition, so it is worth a try.
- Used in production when a category changes very frequently.



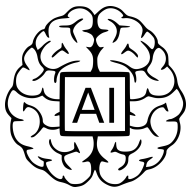
Hashing Encoder

- Converts categorical variables to a higher dimensional space of integers.
- Maintains the distance between two vectors of categorical variables
- The number of dimensions will be far less than One Hot Encoding
- Data are transformed into fewer features - information loss.
- Different categorical values could be represented by the same Hash values.
- Many Kaggle competitors use Hash Encoding to win the competition, so it is worth a try.
- Used in production when a category changes very frequently.



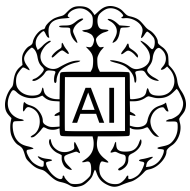
Hashing Encoder

- Converts categorical variables to a higher dimensional space of integers.
- Maintains the distance between two vectors of categorical variables
- The number of dimensions will be far less than One Hot Encoding
- Data are transformed into fewer features - information loss.
- Different categorical values could be represented by the same Hash values.
- Many Kaggle competitors use Hash Encoding to win the competition, so it is worth a try.
- Used in production when a category changes very frequently.



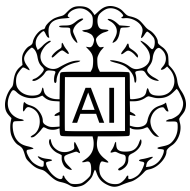
Hashing Encoder

- Converts categorical variables to a higher dimensional space of integers.
- Maintains the distance between two vectors of categorical variables
- The number of dimensions will be far less than One Hot Encoding
- Data are transformed into fewer features - information loss.
- Different categorical values could be represented by the same Hash values.
- Many Kaggle competitors use Hash Encoding to win the competition, so it is worth a try.
- Used in production when a category changes very frequently.



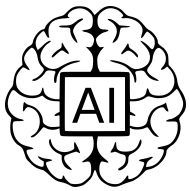
Hashing Encoder

- Converts categorical variables to a higher dimensional space of integers.
- Maintains the distance between two vectors of categorical variables
- The number of dimensions will be far less than One Hot Encoding
- Data are transformed into fewer features - information loss.
- Different categorical values could be represented by the same Hash values.
- Many Kaggle competitors use Hash Encoding to win the competition, so it is worth a try.
- Used in production when a category changes very frequently.



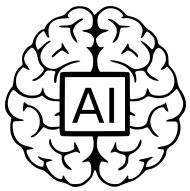
Hashing Encoder


```
hencoder =
    HashingEncoder(cols='Temperature',n_components=3)
hash_res = hencoder.fit_transform(df['Temperature'])
hash_res.sample(5)
```



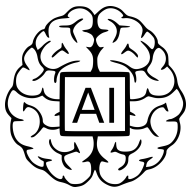
Generalized Linear Mixed Model (GLMM) Encoder

- Similar to TargetEncoder or M-Estimate Encoder.
- Solid statistical theory behind the technique.
- No hyper-parameters to tune. The amount of shrinkage is automatically determined through the estimation process.
- The less observations a category has and/or the more the outcome varies for a category then the higher the regularization towards "the prior" or "grand mean".
- You can just regard this method as applying "GLMM linear regression" on target encoding.
- More time-consuming compare with other methods.



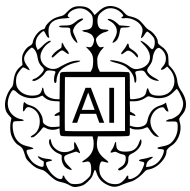
Generalized Linear Mixed Model (GLMM) Encoder

- Similar to TargetEncoder or M-Estimate Encoder.
- Solid statistical theory behind the technique.
- No hyper-parameters to tune. The amount of shrinkage is automatically determined through the estimation process.
- The less observations a category has and/or the more the outcome varies for a category then the higher the regularization towards "the prior" or "grand mean".
- You can just regard this method as applying "GLMM linear regression" on target encoding.
- More time-consuming compare with other methods.



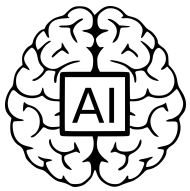
Generalized Linear Mixed Model (GLMM) Encoder

- Similar to TargetEncoder or M-Estimate Encoder.
- Solid statistical theory behind the technique.
- No hyper-parameters to tune. The amount of shrinkage is automatically determined through the estimation process.
- The less observations a category has and/or the more the outcome varies for a category then the higher the regularization towards "the prior" or "grand mean".
- You can just regard this method as applying "GLMM linear regression" on target encoding.
- More time-consuming compare with other methods.



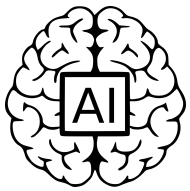
Generalized Linear Mixed Model (GLMM) Encoder

- Similar to TargetEncoder or M-Estimate Encoder.
- Solid statistical theory behind the technique.
- No hyper-parameters to tune. The amount of shrinkage is automatically determined through the estimation process.
- The less observations a category has and/or the more the outcome varies for a category then the higher the regularization towards "the prior" or "grand mean".
- You can just regard this method as applying "GLMM linear regression" on target encoding.
- More time-consuming compare with other methods.



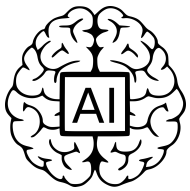
Generalized Linear Mixed Model (GLMM) Encoder

- Similar to TargetEncoder or M-Estimate Encoder.
- Solid statistical theory behind the technique.
- No hyper-parameters to tune. The amount of shrinkage is automatically determined through the estimation process.
- The less observations a category has and/or the more the outcome varies for a category then the higher the regularization towards "the prior" or "grand mean".
- You can just regard this method as applying "GLMM linear regression" on target encoding.
- More time-consuming compare with other methods.

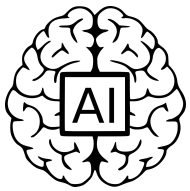


Generalized Linear Mixed Model (GLMM) Encoder

- Similar to TargetEncoder or M-Estimate Encoder.
- Solid statistical theory behind the technique.
- No hyper-parameters to tune. The amount of shrinkage is automatically determined through the estimation process.
- The less observations a category has and/or the more the outcome varies for a category then the higher the regularization towards "the prior" or "grand mean".
- You can just regard this method as applying "GLMM linear regression" on target encoding.
- More time-consuming compare with other methods.



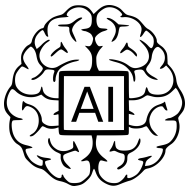

```
glmm_encoder = GLMMEncoder(cols=["month"],
    binomial_target=True)
# binomial_target = True (for Classification)
# binomial_target = False (for Regression)
glmm_encoder.fit(train["month"], target)
X_encoded = glmm_encoder.transform(train["month"])
```



Conclusion

It is essential to understand that all these encodings do not work well in all situations or for every dataset for all machine learning models. Data Scientists still need to experiment and find out which works best for their specific case. If test data has different classes, some of these methods won't work as features won't be similar. There are few benchmark publications by research communities, but it's not conclusive which works best.

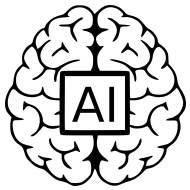
My recommendation will be to try each of these with the smaller datasets and then decide where to focus on tuning the encoding process.



Cross-Validation

Label	train	val	test
OrdinalEncoder	0.85611	0.76588	0.76459
WOEEncoder	0.86433	0.83322	0.77296
TargetEncoder	0.86220	0.82985	0.77866
MEstimateEncoder	0.86547	0.83427	0.77057
JamesSteinEncoder	0.86547	0.83469	0.76749
LeaveOneOutEncoder	1.00000	1.00000	0.51884
CatBoostEncoder	0.82644	0.78127	0.78850
HashingEncoder	0.63969	0.62233	0.61618
OneHotEncoder	0.85759	0.77439	0.77330

Table: XGB CV



The End

Thank you for your attention and patience.