예측모형에서 범주형 변수 처리

1. Dummy variable

   K개의 level이 있는 경우에 (K-1)개의 dummy variable을 생성.

   예를들어 3개의 level A,B,C가 있다고 하자. 그러면 V1, V2 2개의 dummy variable 만들어서 다음과 같이 처리한다

	V1	V2
A	0	0
B	1	0
C	0	1

   레벨의 수보다 1개 적게 만드는 이유는 linear model에서 X-matrix가 LIN을 만족시키기 위해서이다. 선형예측모형에서 모형을 설명할 때 첫번째 레벨의 계수는 0임을 알아야한다. 일반적으로 레벨의 순서는 alphabetical order이다.

2. One-hot-encoding

   K개의 level이 있는 경우에 K개의 dummy variable을 생성. 위와 같은 예제에 다음과 같은 변수를 생성한다.

	V1	V2	V3
A	1	0	0
B	0	1	0
C	0	0	1

   선형예측모형을 사용하면 계수 추정이 안되는 경우가 있다. X-matrix가 Linearly Dependent가 되기 때문에. Tree-based bagging, boosting에서는 아무 문제없음.

3. Impact encoding

   각 범주(level)에 대해 **목표 변수의 평균 값**(또는 다른 통계적 값)을 계산하여 그 값을 범주의 대표값으로 사용

범주형 변수 (`Category`)	**목표 변수 (`Target`)**
A	1
B	0
A	1
C	0
B	1
A	0

   - 목표 변수의 평균을 각 범주별로 계산합니다:
     - `A`: $\frac{1+1+0}{3} = 0.67$
     - `B`: $\frac{0+1}{2} = 0.5$
     - `C`: $\frac{0}{1} = 0$
   - 이를 바탕으로 범주형 변수를 다음과 같이 수치형 변수로 변환합니다:

범주형 변수 (`Category`)	**Impact Encoding (`Category_mean`)**
A	0.67
B	0.5
A	0.67
C	0
B	0.5
A	0.67

   Impact Encoding의 장점

   a. 모델 성능 향상:

   범주형 변수가 많거나, high-cardinality일 때(범주가 매우 많을 때) 유용합니다.

   단순한 one-hot encoding에 비해 메모리와 계산 자원을 절약할 수 있습니다.

   b. 목표 변수와의 관계를 반영:

   범주형 값이 목표 변수와 얼마나 강하게 연관되어 있는지를 직접적으로 반영하기 때문에, 모델 학습에 더 유용한 정보를 제공합니다.

c. 고차원 문제 해결:

One-hot encoding은 범주 수가 많을 경우(고차원 문제) 차원의 저주(curse of dimensionality)를 초래할 수 있지만, Impact Encoding은 이를 완화합니다.

Impact Encoding의 단점

a. 데이터 누출 위험:

목표 변수의 정보를 포함하므로, 훈련 데이터의 통계 정보가 테스트 데이터로 누출될 가능성이 있습니다. 이를 방지하기 위해 K-fold Cross Validation Encoding이나 Out-of-Fold Encoding 같은 기법을 사용합니다.

b. 노이즈 문제:

데이터가 적거나 특정 범주의 샘플이 적을 경우(저빈도 범주), 노이즈가 포함될 가능성이 있습니다. 이를 방지하기 위해 스무딩(Smoothing) 기법을 적용하여 각 범주의 값이 더 안정적으로 계산되도록 합니다.

스무딩(Smoothing) 적용

스무딩 기법은 Impact Encoding에서 빈도가 낮은 범주에 대해 목표 변수의 전체 평균 값을 반영하도록 조정하는 방법입니다.

스무딩 계산식:

$$Impact\_Encoding = \frac{n \cdot Category\_Mean + k \cdot Global\_Mean}{n + k}$$

- n: 해당 범주의 샘플 수
- $Category\_Mean$: 해당 범주의 목표 변수 평균
- $Global\_Mean$: 전체 데이터의 목표 변수 평균
- k: 스무딩 강도를 조정하는 하이퍼파라미터

Out-of-Fold Impact Encoding

훈련 데이터에서 데이터 누출 문제를 방지하기 위해 Out-of-Fold Impact Encoding을 사용합니다:

훈련 데이터를 K개의 Fold로 나눕니다.

각 Fold의 값을 계산할 때, 해당 Fold의 데이터를 제외한 나머지 데이터를 사용해 평균 값을 계산합니다.

이렇게 생성된 값을 각 Fold에 할당합니다.

4. Target encoding(catboost)

   기본적으로 impact encoding기반인데 정보누출 문제를 방지하기 위해서 순차적으로 계산한다. 예를 들어, 한 데이터 포인트의 인코딩 값을 계산할 때 해당 데이터 포인트를 제외한 이전 데이터의 통계를 사용한다.

5. Embedding vector(word2vec)

   Embedding Vector는 각 범주를 고유한 **저차원 벡터 공간의 점**으로 매핑하는 방식입니다. 범주형 변수를 고차원 이진(one-hot) 벡터로 표현하는 대신, 더 작은 차원의 실수 벡터로 변환하여 효율적이고 유용한 정보를 담을 수 있습니다.

   간단한 예:

   범주형 변수 `Category`가 다음과 같다고 가정합시다:

Category
A
B
C

   - **One-Hot Encoding**:
     - A: [1, 0, 0]
     - B: [0, 1, 0]
     - C: [0, 0, 1]
     - 차원: 3
   - **Embedding Vector** (저차원 밀집 벡터):
     - A: [0.12, -0.4]
     - B: [0.5, 0.3]
     - C: [-0.8, 0.1]
     - 차원: 2 (임베딩 차원)

   물론 임베딩 차원은 임의로 설정 가능. 자연어 처리 방법론에서 word2vec 기법도 embedding vector 기법중에 하나임.

Embedding Vector의 특징

1. **저차원 벡터 표현**:
   - 범주형 변수의 고차원 표현(one-hot)을 저차원 벡터로 변환하여 메모리와 계산 효율성을 높임.
   - 임베딩 차원은 데이터 특성에 따라 사용자가 지정(예: 8, 16, 32 등).
2. **정보 압축 및 의미적 유사성**:
   - 학습된 임베딩은 같은 범주 간의 관계를 반영할 수 있습니다.
   - 예: "A"와 "B"가 비슷한 행동을 하면 임베딩 벡터 공간에서 가까운 점으로 매핑됩니다.
3. **학습 기반 인코딩**:
   - 임베딩 벡터는 모델 학습 중에 최적화됩니다.
   - 데이터와 목표 변수 간의 관계를 반영한 벡터로 업데이트됩니다.

Embedding Vector의 학습 방식

1. **임베딩 레이어 초기화**:
   - 각 범주는 고유한 인덱스를 할당받습니다.
   - 초기에는 무작위 값으로 설정된 벡터를 사용합니다.
2. **학습 과정에서 최적화**:
   - 딥러닝 모델의 학습 과정에서 임베딩 벡터도 다른 파라미터처럼 경사 하강법으로 최적화됩니다.
   - 모델이 범주형 변수와 목표 변수 간의 관계를 학습하면서 벡터가 업데이트됩니다.
3. **결과**:
   - 학습이 완료되면, 각 범주는 학습된 밀집 벡터로 표현됩니다.

Embedding Vector와 One-Hot Encoding 비교

특징	Embedding Vector	One-Hot Encoding
차원 수	사용자 정의(저차원, 예: 16, 32 등)	범주 수와 동일
메모리 효율성	메모리 효율적	메모리 비효율적
범주 관계 반영	임베딩 벡터 간 유사성을 반영	범주 간 관계 없음
학습	모델 학습 중에 최적화됨	고정된 인코딩 방식
적합성	고차원, 고유 범주가 많은 데이터에 적합	저차원, 범주가 적은 경우 적합

장점과 단점

장점

1. **저차원 표현**:
   - 고차원 데이터를 효율적으로 압축하여 계산 속도와 메모리 사용량 개선.
2. **관계 학습**:
   - 범주형 변수 간의 유사성을 학습하여 더 나은 모델 성능.
3. **고차원 데이터 처리**:
   - 범주가 많은 경우에도 효과적으로 처리 가능.

단점

1. **학습 필요**:
   - 임베딩 벡터는 학습 과정 중에 최적화되므로, 초기 학습 단계에서는 성능이 낮을 수 있음.
2. **해석 어려움**:
   - 학습된 임베딩 벡터는 모델의 내부 표현으로, 직관적으로 해석하기 어려움.
3. **소량 데이터에서 비효율적**:
   - 데이터가 적을 경우 임베딩 벡터 학습이 어려움.

예제 코드

```
library(keras)
# 데이터 생성
data <- data.frame(
  Category = c("A", "B", "C", "A", "B", "C", "A", "B"),
  Target = c(1, 0, 1, 1, 0, 0, 1, 0)
)
# 범주형 변수를 인덱스로 변환
category_index <- as.integer(as.factor(data$Category)) - 1  # 인덱스는 0부터 시작
target <- data$Target
# 임베딩 차원 설정
num_categories <- length(unique(category_index))
embedding_dim <- 2  # 임베딩 차원
# 모델 정의
input <- layer_input(shape = 1)
embedding <- layer_embedding(input_dim = num_categories, output_dim = embedding_dim)(input)
flatten <- layer_flatten()(embedding)
output <- layer_dense(units = 1, activation = "sigmoid")(flatten)
model <- keras_model(inputs = input, outputs = output)

# 모델 컴파일
model %>% compile(
  optimizer = "adam",
  loss = "binary_crossentropy",
  metrics = c("accuracy")
)

# 모델 학습
model %>% fit(
  x = category_index,
  y = target,
  epochs = 10,
  verbose = 1
)

embedding_weights <- get_weights(model)[[1]]
print(embedding_weights)
```